

Security Assessment Report

Raydium CLMM Incremental

Limit Order and Dynamic Fee

April 21, 2026

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the limit order and the dynamic fee related changes in the Raydium CLMM program.

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

The initial audit focused on the following versions and revealed 11 issues or questions.

Task	Type	Commit
feat_limitorder_dynamicfee diff	Solana	c678919 → f1ec7dc

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

Table of Contents

Result Overview 3
Findings in Detail 4
[M-01] Limit order only match can strand the LP fee remainder 4
[M-02] Shared partial-fill accounting allows over-settlement across orders 5
[L-01] Missing `check_limit_order_amount` in `increase_limit_order` 10
[L-02] Zero amount final settlement blocks `close_limit_order` 12
[L-03] Exact-output swaps can revert when `sqrt_price_limit_x64=0` 14
[L-04] Small remaining exact-input amount can revert limit-order matching 26
[L-05] Legacy swap fails when pool price remains unchanged 28
[I-01] `settle_filled_order` output loses precision on repeated partial settlements 30
[I-02] Base fee rate not capped at `MAX_FEE_RATE_NUMERATOR` in static-fee pools 31
[I-03] `DynamicFeeConfig` validated with hardcoded `tick_spacing` 32
[I-04] Decrease rebasing can discard deferred settlement dust 34
Appendix: Methodology and Scope of Work 38

Result Overview

Issue	Impact	Status
FEAT_LIMITORDER_DYNAMICFEE DIFF		
[M-01] Limit order only match can strand the LP fee remainder	Medium	Resolved
[M-02] Shared partial-fill accounting allows over-settlement across orders	Medium	Resolved
[L-01] Missing `check_limit_order_amount` in `increase_limit_order`	Low	Resolved
[L-02] Zero amount final settlement blocks `close_limit_order`	Low	Resolved
[L-03] Exact-output swaps can revert when `sqrt_price_limit_x64=0`	Low	Resolved
[L-04] Small remaining exact-input amount can revert limit-order matching	Low	Resolved
[L-05] Legacy swap fails when pool price remains unchanged	Low	Resolved
[I-01] `settle_filled_order` output loses precision on repeated partial settlements	Info	Acknowledged
[I-02] Base fee rate not capped at `MAX_FEE_RATE_NUMERATOR` in static-fee pools	Info	Acknowledged
[I-03] `DynamicFeeConfig` validated with hardcoded `tick_spacing`	Info	Acknowledged
[I-04] Decrease rebasing can discard deferred settlement dust	Info	Acknowledged

Findings in Detail

FEAT_LIMITORDER_DYNAMICFEE DIFF

[M-01] Limit order only match can strand the LP fee remainder

Identified in commit `c678919`.

During a swap, fee amounts are generated both from LP-driven swaps and from limit order matches. These fees are split into `protocol fee`, `fund fee`, and `lp fee` via `SwapState::split_fees`. However, the `lp fee` portion is only accumulated when `state.liquidity > 0`.

```
/* programs/amm/src/instructions/swap.rs */
249 | // Update global fee tracker
250 | if self.liquidity > 0 {
251 |     let fee_growth_global_x64_delta = U128::from(remaining_fee)
252 |         .mul_div_floor(U128::from(fixed_point_64::Q64), U128::from(self.liquidity))
253 |         .ok_or(ErrorCode::CalculateOverflow)?
254 |         .as_u128();
255 |
256 |     self.fee_growth_global_x64 = self
257 |         .fee_growth_global_x64
258 |         .checked_add(fee_growth_global_x64_delta)
259 |         .ok_or(ErrorCode::CalculateOverflow)?;
260 |     self.lp_fee = self
261 |         .lp_fee
262 |         .checked_add(remaining_fee)
263 |         .ok_or(ErrorCode::CalculateOverflow)?;
264 | }
```

The introduction of limit orders allows the current tick to match limit orders (`orders_amount > 0`) even when `state.liquidity == 0`. In this scenario, the portion of `fee_amount` that would ordinarily be attributed to LPs (i.e., `100% - protocol_fee_rate - fund_fee_rate`) is neither distributed to LPs, the protocol, the fund, nor limit-order makers — it accumulates in the vault indefinitely with no claimant.

It is recommended that, in cases where `state.liquidity == 0`, the LP fee portion could be redistributed proportionally to the `protocol fee` and `fund fee` buckets, or handled through another suitable distribution mechanism, to prevent these tokens from becoming permanently stranded.

Resolution

Resolved by commit 27c9252.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[M-02] Shared partial-fill accounting allows over-settlement across orders

Identified in commit `c678919`.

The `decrease_limit_order` instruction mutates the shared partially-filled cohort state after calling `settle_filled_order`. Because `settle_filled_order` derives each order's fill amount from the shared ratio `part_filled_orders_remaining / part_filled_orders_total`, a partially filled order can change the fill ratio observed by every other unsettled order at the same tick. This breaks the accounting invariant that the cohort's cumulative settled amount must not exceed the quantity actually matched by `swap`/`swap_v2`.

```
/* programs/amm/src/states/limit_order.rs */
070 | pub fn settle_filled_order(&mut self, tick_state: &TickState) -> Result<u64> {
076 |     let filled_amount = if self.order_phase == tick_state.order_phase {
077 |         // same phase, unfilled - no output // N: checked
078 |         0
079 |     } else if self.order_phase + 1 == tick_state.order_phase {
083 |         // Use ceil to avoid over-attributing fills (protocol safety: don't over-pay output).
084 |         // new_remaining_amount = ceil(total * part_remaining / part_total) = conservative estimate.
             // @audit: Every order in the cohort settles against the same mutable ratio.
085 |         let new_remaining_amount = U128::from(self.total_amount)
086 |             .mul_div_ceil(
087 |                 U128::from(tick_state.part_filled_orders_remaining),
088 |                 U128::from(tick_state.part_filled_orders_total),
089 |             ) // N: remaining, so direction correct
090 |             .ok_or(ErrorCode::CalculateOverflow)?
091 |             .as_u64();
092 |         // filled_amount = remaining - new_remaining (amount matched in this part-fill)
093 |         remaining_amount.saturating_sub(new_remaining_amount)
094 |     } else if self.order_phase + 2 <= tick_state.order_phase {
095 |         // full filled
096 |         remaining_amount
097 |     } else {
098 |         return err!(ErrorCode::InvalidOrderPhase);
099 |     };
100 |
101 |     self.filled_amount = self
102 |         .filled_amount
103 |         .checked_add(filled_amount)
104 |         .ok_or(ErrorCode::CalculateOverflow)?;
105 |
106 |     // Calculate the amount of tokens received after selling via limit order
107 |     let amount_out =
108 |         TickState::get_limit_order_output(filled_amount, tick_state.tick, self.zero_for_one)?;
109 |     Ok(amount_out)
110 | }

/* programs/amm/src/states/limit_order.rs */
131 | pub fn decrease_amount(
132 |     &mut self,
133 |     tick_state: &mut TickState,
134 |     amount: u64,
```

```
135 | ) -> Result<DecreaseAmountResult> {
136 |     let settled_output_amount = self.settle_filled_order(tick_state)?;
137 |     let unfilled_amount = self.get_unfilled_amount()?;
144 |     let real_decrease_amount = amount.min(unfilled_amount);
145 |     if self.order_phase == tick_state.order_phase {
151 |     } else if self.order_phase + 1 == tick_state.order_phase {
            // @audit: This lets one order rewrite the ratio used by every other unsettled order at the tick.
152 |         tick_state.part_filled_orders_remaining = tick_state
153 |             .part_filled_orders_remaining
154 |             .checked_sub(real_decrease_amount)
155 |             .ok_or(ErrorCode::CalculateOverflow)?;
156 |         tick_state.part_filled_orders_total = tick_state
157 |             .part_filled_orders_total
158 |             .checked_sub(real_decrease_amount)
159 |             .ok_or(ErrorCode::CalculateOverflow)?;
160 |     } else {
161 |         return err!(ErrorCode::OrderAlreadyFilled);
162 |     };
190 | }
```

During a partial fill, the entire tick cohort is moved into a shared bucket. All orders that were active at that tick phase then settle against the same `part_filled_orders_remaining / part_filled_orders_total` ratio.

```
/* programs/amm/src/states/tick_array.rs */
566 | // Order phase increases when consuming from orders_amount
567 | self.order_phase = self.order_phase.saturating_add(1);
568 |
569 | // Set part_filled_orders_total to the original orders_amount
570 | // This represents the total amount that was moved to partial fill state
571 | self.part_filled_orders_total = self.orders_amount;
572 |
573 | // Move remaining orders_amount to part_filled_orders_remaining
574 | self.part_filled_orders_remaining = self
575 |     .part_filled_orders_remaining
576 |     .saturating_add(self.orders_amount - amount_out_continue_to_consume);
```

This allows the following instruction-level attack:

1. Call `open_limit_order` twice to create orders A and B at the same tick, in the same direction, and therefore in the same `order_phase`, each with `total_amount = 1000`. `open_limit_order` stores the current `tick_state.order_phase` into each order and adds both amounts to `tick_state.orders_amount`.
2. Trigger a partial match at that tick with `swap` or `swap_v2` in the opposite direction. When only `1000` of the `2000` total order amount is consumed, the swap path advances `tick_state.order_phase` and moves the cohort into the shared partial-fill bucket with `part_filled_orders_total = 2000` and `part_filled_orders_remaining = 1000`.
3. Call `decrease_limit_order` on order A with `amount = 500`. This instruction invokes `LimitOrderState::decrease_amount`, which first calls `settle_filled_order`, so order A receives the correct output for

a `500` fill. The same instruction then subtracts `500` from both `part_filled_orders_remaining` and `part_filled_orders_total`, leaving the shared cohort state as `part_filled_orders_remaining = 500` and `part_filled_orders_total = 1500`, while also refunding A's `500` unfilled input amount.

4. Call `settle_limit_order` on order B. Because B now settles against the distorted shared ratio `500 / 1500`, it computes `new_remaining = ceil(1000 * 500 / 1500) = 334` and records `filled_amount = 666` instead of `500`.
5. Only `1000` units were actually matched by the `swap`/`swap_v2` instruction, but the two orders have already claimed output corresponding to `500 + 666 = 1166` units of fill. The cohort has therefore over-settled by `166` units relative to the actual match quantity.
6. The attacker can then withdraw the remaining unfilled input from the outstanding order(s).

Settled output is transferred directly from the pool vault, so the inflated `filled_amount` turns into a real overpayment of output tokens relative to the tick's true matched amount. However, when the same actor controls both the resting orders and the opposing swap, this loop is not automatically a risk-free profit in mark-to-market terms: in the concrete example below, the attacker ends with `166` extra units of output but also leaves `166` extra units of input in the pool. At the limit price, this is economically similar to forcing an extra `166` units of execution rather than minting free value. Even so, the protocol's settlement accounting is still broken because:

1. The cohort settles more filled quantity than the swap actually consumed,
2. Later orders observe a manipulated ratio and receive the wrong pro-rata result, and
3. The pool's vault balances are moved according to a corrupted state rather than the real execution.

If the current price in the pool is relatively low, the attacker might be motivated to force this extra swap.

Meanwhile, under normal circumstances, the funds of all limit orders are completely independent of the funds corresponding to the liquidity. All tokens originate from the input when opening a limit order and the input when a swap matches the limit order. Theoretically, this process will not affect the amounts and ratios of the two tokens corresponding to the liquidity.

However, due to the over-settle issue mentioned above, the output tokens generated without a real swap actually come from the liquidity funds. Therefore, this will lead to the misappropriation of the funds corresponding to the liquidity.

Subsequently, if all partially fulfilled orders on this tick are completely consumed immediately afterward, the pool will return to the correct state. But if the price and tick move in the opposite direction of the current order and leave these partially fulfilled orders behind, this token misappropriation will

persist. Consequently, all non-active ticks might contain these misappropriated tokens. If this occurs consistently in a single direction, it could lead to extreme situations where the tokens in a specific direction are insufficient to cover the tokens required for decreasing liquidity, ultimately causing a DoS.

Any legitimate user who decreases a partially-filled order triggers the same distortion for all other unsettled orders at that tick. This issue is not limited to attackers. It happens during normal operations and slowly accumulates composition errors.

It is recommended to ensure that one partially filled order cannot change the settlement ratio seen by other orders in the same cohort. The safest fixes are to either disallow `decrease_amount` once an order has entered the partially-filled phase, or to snapshot immutable per-order settlement data when the tick advances and settle each order against that immutable state.

PoC:

```
#[test]
fn partial_fill_decrease_distorts_other_orders_poc_test() {
    let tick = 0;
    let initial_order_phase = 0;
    let limit_zero_for_one = true;

    let mut tick_state = create_mock_tick_state(tick, initial_order_phase, 0, 0, 0);

    let mut order_a = open_order(
        1000,
        limit_zero_for_one,
        tick_state.order_phase,
        &mut tick_state,
    );
    let mut order_b = open_order(
        1000,
        limit_zero_for_one,
        tick_state.order_phase,
        &mut tick_state,
    );

    // Consume half of the shared cohort. Both orders move into the partially-filled phase.
    tick_state
        .match_limit_order(1000, !limit_zero_for_one, false, 0, true)
        .unwrap();
    let order_phase = tick_state.order_phase;
    let part_filled_orders_total = tick_state.part_filled_orders_total;
    let part_filled_orders_remaining = tick_state.part_filled_orders_remaining;
    assert_eq!(order_phase, initial_order_phase + 1);
    assert_eq!(part_filled_orders_total, 2000);
    assert_eq!(part_filled_orders_remaining, 1000);

    // Decreasing A now mutates the shared partial-fill ratio for every other unsettled order.
    let decrease_a = order_a.decrease_amount(&mut tick_state, 500).unwrap();
    assert_eq!(decrease_a.settled_output_amount, 500);
    assert_eq!(decrease_a.real_decrease_amount, 500);
    assert_eq!(order_a.total_amount, 500);
```

```
    assert_eq!(order_a.filled_amount, 500);
    let part_filled_orders_total = tick_state.part_filled_orders_total;
    let part_filled_orders_remaining = tick_state.part_filled_orders_remaining;
    assert_eq!(part_filled_orders_total, 1500);
    assert_eq!(part_filled_orders_remaining, 500);

    // Order B now settles against the distorted ratio and receives 666 instead of 500.
    let settle_b_after_a_decrease = order_b.settle_filled_order(&tick_state).unwrap();
    assert_eq!(settle_b_after_a_decrease, 666);
    assert_eq!(order_b.filled_amount, 666);

    // Only 1000 units were actually matched, but the two orders have now claimed 1166.
    assert_eq!(decrease_a.settled_output_amount + settle_b_after_a_decrease, 1166);
}
```

Resolution

Resolved by commit f1ec7dc.

This issue was fixed by changing limit-order settlement from a shared mutable cohort-based calculation to an order-local segmented accounting model. The patch introduces per-order settlement state, such as `settle_base`, `settled_output`, and an order-level `unfilled_ratio_x64` snapshot, so each order now settles against its own fixed segment baseline and only receives the delta between cumulative entitled output and previously paid output. As a result, `decrease_amount` only resets the calling order's settlement segment and no longer changes the settlement ratio observed by other partially filled orders at the same tick.

This design and accounting approach does not accumulate error on each `decrease` or `settle`. However, if a `decrease` occurs after a non-exact partial match, due to rounding, `settle_base` may be reset to a slightly understated value, which can cause the user to lose up to one unit of input token. We consider this an acceptable form of bounded dust loss. Please see [I-04] for details.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[L-01] Missing `check_limit_order_amount` in `increase_limit_order`

Identified in commit `c678919`.

The `open_limit_order` instruction validates that a limit order's implied output amount falls within the safe range `[1, u64::MAX - 2]` by calling `check_limit_order_amount` before the order is placed:

```
/* programs/amm/src/instructions/limit_order/open_limit_order.rs */
210 | /// (at this tick's price) must be in [1, u64::MAX - 2].
211 | fn check_limit_order_amount(amount: u64, tick_index: i32, zero_for_one: bool) -> Result<()> {
228 |     if amount_out < U128::one() || amount_out >= U128::from(u64::MAX - 2) {
229 |         return err!(ErrorCode::InvalidLimitOrderAmount);
230 |     }
231 |     Ok(())
232 | }
```

However, the `increase_limit_order` instruction does not re-run this guard after mutating `total_amount` and `tick_state.orders_amount`:

```
/* programs/amm/src/instructions/limit_order/increase_limit_order.rs */
059 | pub fn increase_limit_order(ctx: Context<IncreaseLimitOrder>, amount: u64) -> Result<()> {
060 |     require!(amount > 0, ErrorCode::ZeroAmountSpecified);
077 |     ctx.accounts
078 |         .limit_order
079 |         .increase_amount(tick_state, amount_without_transfer_fee)?;
```

As a result, the limit order will be treated as a valid order, but the amount will be out of range, thus causing the swap to fail.

```
/* programs/amm/src/states/tick_array.rs */
435 | let amount_in = if zero_for_one {
436 |     let token_0_price_x64 = tick_math::get_price_at_tick(tick, true)?;
437 |     // token1_consumed = token0_executed * token_0_price_x64 / 2^64
438 |     U128::from(amount_out)
439 |         .mul_div_ceil(token_0_price_x64, U128::from(fixed_point_64::Q64))
440 |         .ok_or(ErrorCode::CalculateOverflow)?
441 |         .as_u64()
442 | } else {
443 |     let token_0_price_x64 = tick_math::get_price_at_tick(tick, false)?;
444 |     // token0_consumed = token1_executed * 2^64 / token_0_price_x64
445 |     U128::from(amount_out)
446 |         .mul_div_ceil(U128::from(fixed_point_64::Q64), token_0_price_x64)
447 |         .ok_or(ErrorCode::CalculateOverflow)?
448 |         .as_u64()
```

Consider adding `check_limit_order_amount` in `increase_amount` before updating `total_amount`.

Resolution

Resolved by commit 27c9252.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[L-02] Zero amount final settlement blocks `close_limit_order`

Identified in commit `c678919`.

The `settle_filled_order` instruction requires that the output token amount generated by the settle process is not zero. However, because this instruction can be called multiple times and each call performs a partial fill, and the final calculation result is rounded down, an issue may occur after a user settles multiple times. Even if the corresponding `tick` and `order_phase` are already fulfilled, the final remaining input tokens of the current order might not be enough to swap for one unit of output token. Under this circumstance, it is impossible to set the `unfilled_amount` (`total_amount - filled_amount`) in the `LimitOrderState` to zero through the `settle_limit_order` instruction.

```
/* programs/amm/src/instructions/limit_order/settle_limit_order.rs */
057 | pub fn settle_limit_order(ctx: Context<SettleLimitOrder>) -> Result<()> {
065 |     let amount_out = order.settle_filled_order(tick_state)?;
066 |     require_gt!(amount_out, 0);
092 | }
```

Another instruction that reduces the `unfilled_amount` is `decrease_limit_order`, but it cannot be used when the `order_phase` corresponding to the order has already been fulfilled.

```
/* programs/amm/src/states/limit_order.rs */
131 | pub fn decrease_amount(
132 |     &mut self,
133 |     tick_state: &mut TickState,
134 |     amount: u64,
135 | ) -> Result<DecreaseAmountResult> {
145 |     if self.order_phase == tick_state.order_phase {
    ...
151 |     } else if self.order_phase + 1 == tick_state.order_phase {
    ...
160 |     } else {
161 |         return err!(ErrorCode::OrderAlreadyFilled);
162 |     };
190 | }
```

This will prevent the user from calling `close_limit_order` later, making it impossible to recover the account rent.

We recommend that during `settle_limit_order`, if the current settlement has already set the `unfilled_amount` of the current order to zero, which means the order is fulfilled, the protocol should no longer require `amount_out` to be greater than zero.

Resolution

Resolved by commit 27c9252.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[L-03] Exact-output swaps can revert when `sqrt_price_limit_x64=0`

Identified in commit `c678919`.

The `swap_v2` instruction supports Token-2022 transfer-fee mints by converting the caller's requested exact-output amount into the gross output amount that must leave the pool vault. This is intended to let the user receive the requested post-fee amount while still using the same `swap_internal` accounting flow.

In the exact-output path, however, the no-partial-fill (`sqrt_price_limit_x64 == 0`) guard still compares the original `amount_specified` value against `transfer_amount_0` or `transfer_amount_1`. In these branches `amount_specified` remains the user's desired net receipt, while `amount_calculate_specified` and the later `transfer_amount_*` values represent the gross Token-2022 transfer amount. When the output mint charges a non-zero transfer fee, those values differ by construction, so the final equality check may reject a fully satisfied swap even though the user would receive the correct net output.

```
/* programs/amm/src/instructions/swap_v2.rs */
096 | let (amount_calculate_specified, transfer_fee) = if is_base_input {
097 |     let transfer_fee =
098 |         util::get_transfer_fee(ctx.input_vault_mint.clone(), amount_specified).unwrap();
099 |     (amount_specified - transfer_fee, transfer_fee)
100 | } else {
101 |     let transfer_fee =
102 |         util::get_transfer_inverse_fee(ctx.output_vault_mint.clone(), amount_specified)
103 |             .unwrap();
104 |     (amount_specified + transfer_fee, transfer_fee)
105 | }; // @audit: if is_base_out, amount_specified should be the net output

/* programs/amm/src/instructions/swap_v2.rs */
311 | if sqrt_price_limit_x64 == 0 {
312 |     // Does't allow partial filled without specified limit_price.
313 |     if is_base_input {
314 |         if zero_for_one {
315 |             require_eq!(amount_specified, transfer_amount_0);
316 |         } else {
317 |             require_eq!(amount_specified, transfer_amount_1);
318 |         }
319 |     } else {
320 |         if zero_for_one {
321 |             require_eq!(amount_specified, transfer_amount_1);
322 |         } else {
323 |             require_eq!(amount_specified, transfer_amount_0);
324 |         }
325 |     }
326 | }
```

The root cause appears to be a mismatch between net-output and gross-transfer domains in the exact-

output branch. This may make exact-output swaps unavailable for transfer-fee output mints whenever callers rely on the default `sqrt_price_limit_x64 == 0` behavior. Consider performing this final equality check against `amount_calculate_specified` in the exact-output branch.

PoC

```
import * as anchor from "@coral-xyz/anchor";
import { Program } from "@coral-xyz/anchor";
import { RaydiumClmm } from "../target/types/raydium_clmm";
import {
  PublicKey,
  Keypair,
  SystemProgram,
  SYSVAR_RENT_PUBKEY,
  AccountMeta,
  ComputeBudgetProgram,
} from "@solana/web3.js";
import {
  TOKEN_PROGRAM_ID,
  TOKEN_2022_PROGRAM_ID,
  createMint,
  createAccount,
  mintTo,
  getAssociatedTokenAddressSync,
  createAssociatedTokenAccountInstruction,
  createAssociatedTokenAccount,
  ASSOCIATED_TOKEN_PROGRAM_ID,
  getAccount,
  ExtensionType,
  createInitializeMintInstruction,
  createInitializeTransferFeeConfigInstruction,
  getMintLen,
} from "@solana/spl-token";
import { assert, expect } from "chai";
import {
  SqrtPriceMath,
  TickUtils,
  MEMO_PROGRAM_ID,
  METADATA_PROGRAM_ID,
} from "@raydium-io/raydium-sdk-v2";
import { PDAUtils } from "./utils/pda";

describe("exact-output swapV2 fails with Token-2022 transfer fee", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.raydiumClmm as Program<RaydiumClmm>;
  const admin = provider.wallet.payer;
  const pda = new PDAUtils(program.programId);

  // Mints
  let token2022Mint: PublicKey; // Token-2022 with transfer fee
  let splMint: PublicKey; // Standard SPL Token
  let token0: PublicKey; // The one with smaller pubkey
  let token1: PublicKey; // The one with larger pubkey
  let token0IsToken2022: boolean;

  // Token programs for token0 and token1
```

```
let tokenProgram0: PublicKey;
let tokenProgram1: PublicKey;

// Token accounts (for admin)
let token0Account: PublicKey;
let token1Account: PublicKey;

// Pool
let ammConfig: PublicKey;
let poolAddress: PublicKey;

// Transfer fee config: 1% (100 bps), max fee = 1_000_000_000
const TRANSFER_FEE_BPS = 100; // 1%
const MAX_FEE = BigInt(1_000_000_000);

/**
 * Create a Token-2022 mint with TransferFeeConfig extension
 */
async function createToken2022MintWithTransferFee(): Promise<PublicKey> {
  const mintKeypair = Keypair.generate();
  const decimals = 9;

  // Calculate space needed for mint with transfer fee extension
  const mintLen = getMintLen([ExtensionType.TransferFeeConfig]);
  const lamports =
    await provider.connection.getMinimumBalanceForRentExemption(mintLen);

  // Build transaction:
  // 1. Create account
  // 2. Initialize transfer fee config
  // 3. Initialize mint
  const transaction = new anchor.web3.Transaction().add(
    SystemProgram.createAccount({
      fromPubkey: admin.publicKey,
      newAccountPubkey: mintKeypair.publicKey,
      space: mintLen,
      lamports,
      programId: TOKEN_2022_PROGRAM_ID,
    }),
    createInitializeTransferFeeConfigInstruction(
      mintKeypair.publicKey,
      admin.publicKey, // transferFeeConfigAuthority
      admin.publicKey, // withdrawWithheldAuthority
      TRANSFER_FEE_BPS, // feeBasisPoints
      MAX_FEE, // maxFee
      TOKEN_2022_PROGRAM_ID
    ),
    createInitializeMintInstruction(
      mintKeypair.publicKey,
      decimals,
      admin.publicKey, // mintAuthority
      null, // freezeAuthority
      TOKEN_2022_PROGRAM_ID
    )
  );

  await provider.sendAndConfirm(transaction, [admin, mintKeypair]);
  return mintKeypair.publicKey;
}
```

```
before(async () => {
  // Airdrop SOL
  const sig = await provider.connection.requestAirdrop(
    admin.publicKey,
    10 * anchor.web3.LAMPORTS_PER_SOL
  );
  await provider.connection.confirmTransaction(sig);

  // 1. Create Token-2022 mint with transfer fee
  token2022Mint = await createToken2022MintWithTransferFee();
  console.log(
    "Token-2022 mint (with transfer fee):",
    token2022Mint.toBase58()
  );

  // 2. Create standard SPL Token mint
  splMint = await createMint(
    provider.connection,
    admin,
    admin.publicKey,
    null,
    9
  );
  console.log("Standard SPL mint:", splMint.toBase58());

  // 3. Order mints: token0 < token1 by pubkey
  if (token2022Mint.toBuffer().compare(splMint.toBuffer()) < 0) {
    token0 = token2022Mint;
    token1 = splMint;
    token0IsToken2022 = true;
  } else {
    token0 = splMint;
    token1 = token2022Mint;
    token0IsToken2022 = false;
  }

  tokenProgram0 = token0IsToken2022
    ? TOKEN_2022_PROGRAM_ID
    : TOKEN_PROGRAM_ID;
  tokenProgram1 = token0IsToken2022
    ? TOKEN_PROGRAM_ID
    : TOKEN_2022_PROGRAM_ID;

  console.log(
    `token0=${token0.toBase58()} (${
      token0IsToken2022 ? "Token-2022" : "SPL"
    })`
  );
  console.log(
    `token1=${token1.toBase58()} (${
      token0IsToken2022 ? "SPL" : "Token-2022"
    })`
  );

  // 4. Create token accounts and mint tokens
  const mintAmount = 1_000_000_000_000; // 1000 tokens (9 decimals)

  // For Token-2022 mint: create ATA under Token-2022 program
  const token2022ATA = await createAssociatedTokenAccount(
    provider.connection,
```

```
    admin,
    token2022Mint,
    admin.publicKey,
    undefined,
    TOKEN_2022_PROGRAM_ID
  );

  // For standard SPL mint: create account under standard Token program
  const splATA = await createAssociatedTokenAccount(
    provider.connection,
    admin,
    splMint,
    admin.publicKey
  );

  // Mint tokens
  await mintTo(
    provider.connection,
    admin,
    token2022Mint,
    token2022ATA,
    admin,
    mintAmount,
    undefined,
    undefined,
    TOKEN_2022_PROGRAM_ID
  );

  await mintTo(
    provider.connection,
    admin,
    splMint,
    splATA,
    admin,
    mintAmount
  );

  // Set token accounts in order
  if (token0IsToken2022) {
    token0Account = token2022ATA;
    token1Account = splATA;
  } else {
    token0Account = splATA;
    token1Account = token2022ATA;
  }

  // 5. Create AMM config
  const ammConfigIndex = 0;
  const [ammConfigAddr] = await pda.getAmmConfigPDA(ammConfigIndex);
  ammConfig = ammConfigAddr;

  try {
    await program.account.ammConfig.fetch(ammConfig);
    console.log("AMM config already exists");
  } catch {
    console.log("Creating AMM config...");
    await program.methods
      .createAmmConfig(
        0,
        10, // tickSpacing
```

```
      2500, // tradeFeeRate: 0.25%
      120000, // protocolFeeRate: 12%
      0 // fundFeeRate
    )
    .accounts({
      admin: admin.publicKey,
      ammConfig: ammConfig,
      systemProgram: SystemProgram.programId,
    })
    .signers([admin])
    .rpc();
  console.log("AMM config created");
}

// 6. Create pool
const startTick = 0;
const sqrtPriceX64 = SqrtPriceMath.getSqrtPriceX64FromTick(startTick);

const [poolAddr] = await pda.getPoolStatePDA(ammConfig, token0, token1);
poolAddress = poolAddr;
console.log("Pool address:", poolAddress.toBase58());

const [tokenVault0] = await pda.getTokenVaultPDA(poolAddress, token0);
const [tokenVault1] = await pda.getTokenVaultPDA(poolAddress, token1);
const [observationState] = await pda.getObservationStatePDA(poolAddress);
const [tickArrayBitmap] = await pda.getTickArrayBitmapPDA(poolAddress);

const openTime = new anchor.BN(Math.floor(Date.now() / 1000) - 100);

await program.methods
  .createPool(sqrtPriceX64, openTime)
  .accounts({
    poolCreator: admin.publicKey,
    ammConfig: ammConfig,
    poolState: poolAddress,
    tokenMint0: token0,
    tokenMint1: token1,
    tokenVault0: tokenVault0,
    tokenVault1: tokenVault1,
    observationState: observationState,
    tickArrayBitmap: tickArrayBitmap,
    tokenProgram0: tokenProgram0,
    tokenProgram1: tokenProgram1,
    systemProgram: SystemProgram.programId,
    rent: SYSVAR_RENT_PUBKEY,
  } as any)
  .signers([admin])
  .rpc();

console.log("Pool created");

// 7. Open position (add liquidity) using openPositionV2
//    Range: tick -100 to +100
const tickLowerIndex = -100;
const tickUpperIndex = 100;
const poolStateData = await program.account.poolState.fetch(poolAddress);
const tickSpacing = poolStateData.tickSpacing;

const lowerStart = TickUtils.getTickArrayStartIndexByTick(
  tickLowerIndex,
```

```
    tickSpacing
  );
  const upperStart = TickUtils.getTickArrayStartIndexByTick(
    tickUpperIndex,
    tickSpacing
  );
  const tickArrayLower = TickUtils.getTickArrayAddressByTick(
    program.programId,
    poolAddress,
    lowerStart,
    tickSpacing
  );
  const tickArrayUpper = TickUtils.getTickArrayAddressByTick(
    program.programId,
    poolAddress,
    upperStart,
    tickSpacing
  );
  const [protocolPosition] = await pda.getProtocolPositionStatePDA(
    poolAddress,
    tickLowerIndex,
    tickUpperIndex
  );

  const positionNftMint = Keypair.generate();
  const positionNftAccount = getAssociatedTokenAddressSync(
    positionNftMint.publicKey,
    admin.publicKey
  );
  const [personalPosition] = await pda.getPersonalPositionStatePDA(
    positionNftMint.publicKey
  );
  const metadataAccount = (
    await PublicKey.findProgramAddress(
      [
        Buffer.from("metadata"),
        METADATA_PROGRAM_ID.toBuffer(),
        positionNftMint.publicKey.toBuffer(),
      ],
      METADATA_PROGRAM_ID
    )
  )[0];

  await program.methods
    .openPositionV2(
      tickLowerIndex, // tickLowerIndex
      tickUpperIndex, // tickUpperIndex
      lowerStart, // tickArrayLowerStartIndex
      upperStart, // tickArrayUpperStartIndex
      new anchor.BN("500000000"), // liquidity
      new anchor.BN("100000000000"), // amount0Max
      new anchor.BN("100000000000"), // amount1Max
      true, // withMetadata
      null // baseFlag
    )
    .accounts({
      payer: admin.publicKey,
      positionNftOwner: admin.publicKey,
      positionNftMint: positionNftMint.publicKey,
      positionNftAccount: positionNftAccount,
```

```
      metadataAccount: metadataAccount,
      poolState: poolAddress,
      protocolPosition: protocolPosition,
      tickArrayLower: tickArrayLower,
      tickArrayUpper: tickArrayUpper,
      personalPosition: personalPosition,
      tokenAccount0: token0Account,
      tokenAccount1: token1Account,
      tokenVault0: tokenVault0,
      tokenVault1: tokenVault1,
      rent: SYSVAR_RENT_PUBKEY,
      systemProgram: SystemProgram.programId,
      tokenProgram: TOKEN_PROGRAM_ID,
      associatedTokenProgram: ASSOCIATED_TOKEN_PROGRAM_ID,
      metadataProgram: METADATA_PROGRAM_ID,
      tokenProgram2022: TOKEN_2022_PROGRAM_ID,
      vault0Mint: token0,
      vault1Mint: token1,
    } as any)
    .preInstructions([
      ComputeBudgetProgram.setComputeUnitLimit({ units: 1400000 }),
    ])
    .signers([admin, positionNftMint])
    .rpc();

  console.log("Position opened with liquidity");

  // Verify pool state
  const poolAfter = await program.account.poolState.fetch(poolAddress);
  console.log("Pool liquidity:", poolAfter.liquidity.toString());
  console.log("Pool tick_current:", poolAfter.tickCurrent);
  console.log("Pool sqrt_price_x64:", poolAfter.sqrtPriceX64.toString());
});

async function doSwapV2(params: {
  inputMint: PublicKey;
  outputMint: PublicKey;
  amount: anchor.BN;
  otherAmountThreshold: anchor.BN;
  sqrtPriceLimitX64: anchor.BN;
  isBaseInput: boolean;
}) {
  const [observationState] = await pda.getObservationStatePDA(poolAddress);
  const [inputVault] = await pda.getTokenVaultPDA(
    poolAddress,
    params.inputMint
  );
  const [outputVault] = await pda.getTokenVaultPDA(
    poolAddress,
    params.outputMint
  );

  // Determine token programs for input/output
  const inputIsToken2022 = params.inputMint.equals(token2022Mint);
  const outputIsToken2022 = params.outputMint.equals(token2022Mint);

  // User token accounts - use the correct program for ATA derivation
  const inputTokenAccount = getAssociatedTokenAddressSync(
    params.inputMint,
    admin.publicKey,
```

```
    false,
    inputIsToken2022 ? TOKEN_2022_PROGRAM_ID : TOKEN_PROGRAM_ID
  );
  const outputTokenAccount = getAssociatedTokenAddressSync(
    params.outputMint,
    admin.publicKey,
    false,
    outputIsToken2022 ? TOKEN_2022_PROGRAM_ID : TOKEN_PROGRAM_ID
  );

  const zeroForOne = params.inputMint.equals(token0);
  const poolStateData = await program.account.poolState.fetch(poolAddress);
  const tickSpacing = poolStateData.tickSpacing;
  const tickCurrent = poolStateData.tickCurrent;
  const ticksPerArray = 60 * tickSpacing; // 600

  // Always include the tick array containing the current tick
  const currentArrayStart = TickUtils.getTickArrayStartIndexByTick(
    tickCurrent,
    tickSpacing
  );

  const tickArrayAddresses: PublicKey[] = [];

  if (zeroForOne) {
    // Going DOWN: include current array and the one below
    tickArrayAddresses.push(
      TickUtils.getTickArrayAddressByTick(
        program.programId,
        poolAddress,
        currentArrayStart,
        tickSpacing
      )
    );
    const belowArrayStart = currentArrayStart - ticksPerArray;
    tickArrayAddresses.push(
      TickUtils.getTickArrayAddressByTick(
        program.programId,
        poolAddress,
        belowArrayStart,
        tickSpacing
      )
    );
  } else {
    // Going UP: current array has tick 100, should be enough
    tickArrayAddresses.push(
      TickUtils.getTickArrayAddressByTick(
        program.programId,
        poolAddress,
        currentArrayStart,
        tickSpacing
      )
    );
  }

  const remainingAccounts: AccountMeta[] = tickArrayAddresses.map(
    (pubkey) => ({
      pubkey,
      isWritable: true,
      isSigner: false,
```

```
    })
  );

  return await program.methods
    .swapV2(
      params.amount,
      params.otherAmountThreshold,
      params.sqrtPriceLimitX64,
      params.isBaseInput
    )
    .accounts({
      payer: admin.publicKey,
      ammConfig: ammConfig,
      poolState: poolAddress,
      inputTokenAccount: inputTokenAccount,
      outputTokenAccount: outputTokenAccount,
      inputVault: inputVault,
      outputVault: outputVault,
      observationState: observationState,
      tokenProgram: TOKEN_PROGRAM_ID,
      tokenProgram2022: TOKEN_2022_PROGRAM_ID,
      memoProgram: MEMO_PROGRAM_ID,
      inputVaultMint: params.inputMint,
      outputVaultMint: params.outputMint,
    } as any)
    .remainingAccounts(remainingAccounts)
    .preInstructions([
      ComputeBudgetProgram.setComputeUnitLimit({ units: 1400000 }),
    ])
    .signers([admin])
    .rpc();
}

it("exact-output swap with sqrtPriceLimitX64=0 FAILS due to require_eq! mismatch", async () => {
  const inputMint = splMint;
  const outputMint = token2022Mint;
  const amount = new anchor.BN(1000); // desired output
  const sqrtPriceLimitX64 = new anchor.BN(0); // zero = no limit = triggers require_eq!

  try {
    await doSwapV2({
      inputMint,
      outputMint,
      amount,
      otherAmountThreshold: new anchor.BN("999999999999"), // max slippage for exact-output
      sqrtPriceLimitX64,
      isBaseInput: false, // exact-output
    });
    assert.fail(
      "Expected exact-output swap with sqrtPriceLimitX64=0 to FAIL, but it succeeded"
    );
  } catch (err: any) {
    // The swap should fail with the require_eq! error
    console.log("EXPECTED FAILURE:", err.message?.substring(0, 200));
    // Anchor wraps the error; check that it's a program error
    assert.ok(
      err.message || err.toString(),
      "Expected an error from the program"
    );
  }
```

```
  });

  it("exact-output swap with non-zero sqrtPriceLimitX64 SUCCEEDS", async () => {
    const inputMint = splMint;
    const outputMint = token2022Mint;
    const amount = new anchor.BN(1000); // desired output

    const zeroForOne = !token0IsToken2022;
    const MIN_SQRT_PRICE_X64_PLUS_1 = new anchor.BN("4295048017");
    const MAX_SQRT_PRICE_X64_MINUS_1 = new anchor.BN(
      "79226673521066979257578248090"
    );
    const sqrtPriceLimitX64 = zeroForOne
      ? MIN_SQRT_PRICE_X64_PLUS_1
      : MAX_SQRT_PRICE_X64_MINUS_1;

    const tx = await doSwapV2({
      inputMint,
      outputMint,
      amount,
      otherAmountThreshold: new anchor.BN("999999999999"), // max slippage for exact-output
      sqrtPriceLimitX64,
      isBaseInput: false, // exact-output
    });

    console.log(
      "BYPASS PASSED: exact-output swap with non-zero sqrtPriceLimitX64 succeeded, tx:",
      tx
    );
  });

});
```

Output:

```
Token-2022 mint (with transfer fee): 3MQbLUAiXy4VhHULDxjAxk6TDToFLMDpCgBetzsoTc5d
Standard SPL mint: CbHqHRxNKqiBMRU1VnVF3kDouME3EkVs4Mu47Gj1qsQr
token0=3MQbLUAiXy4VhHULDxjAxk6TDToFLMDpCgBetzsoTc5d (Token-2022)
token1=CbHqHRxNKqiBMRU1VnVF3kDouME3EkVs4Mu47Gj1qsQr (SPL)
AMM config already exists
Pool address: GUb6KcpVG6a1w31BUwsTtgNiuERHomp9QYJX13NGsjNL
Pool created
Position opened with liquidity
Pool liquidity: 500000000
Pool tick_current: 0
Pool sqrt_price_x64: 18446744073709551616
EXPECTED FAILURE: AnchorError thrown in programs/amm/src/instructions/swap_v2.rs:323. Error Code:
↪  RequireEqViolated. Error Number: 2501. Error Message: A require_eq expression was violated.
Program log: Left: 1000
Prog
      exact-output swap with sqrtPriceLimitX64=0 FAILS due to require_eq! mismatch
BYPASS PASSED: exact-output swap with non-zero sqrtPriceLimitX64 succeeded, tx:
↪  4BdWmSeAELUUUxMoMKCV43M4afBycMS2BvvcaJiUvAB1G3t9dWZsYVjGUvCLcCnYiB7DBPNQgX4WqTDpaXBksZfn
      exact-output swap with non-zero sqrtPriceLimitX64 SUCCEEDS (446ms)
```

Resolution

Resolved by commit 8f0b4be.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[L-04] Small remaining exact-input amount can revert limit-order matching

Identified in commit `c678919`.

The swap loop calls `match_limit_order` with `state.amount_specified_remaining` whenever execution is already sitting on a tick boundary that can be matched through limit orders.

```
/* programs/amm/src/instructions/swap.rs */
617 | let limit_order_unfilled_amount_before =
618 |     next_initialized_tick.limit_order_unfilled_amount()?;
619 | if state.sqrt_price_next_x64 == swap_computed_result.sqrt_price_next_x64 {
620 |     // try to match limit orders on this tick
621 |     let limit_order_result = next_initialized_tick.match_limit_order(
622 |         state.amount_specified_remaining,
623 |         zero_for_one,
624 |         is_base_input,
625 |         total_fee_rate,
626 |         is_fee_on_input,
627 |     )?;
```

In the exact-input branch of match_limit_order, the program derives amount_out from amount_in using the limit-order price and then requires both values to be greater than zero:

```
/* programs/amm/src/states/tick_array.rs */
404 | /// Get the output amount of a limit order
405 | /// amount_in: the amount of the input token
406 | /// zero_for_one: the direction of the input token
407 | /// output amount is rounded down
408 | pub fn get_limit_order_output(amount_in: u64, tick: i32, zero_for_one: bool)

/* programs/amm/src/states/tick_array.rs */
486 | result.amount_out = TickState::get_limit_order_output(
487 |     result.amount_in,
488 |     self.tick,
489 |     swap_direction_zero_for_one,
490 | )?;
542 | require_gt!(result.amount_out, 0, ErrorCode::ZeroAmountSpecified);
543 | require_gt!(result.amount_in, 0, ErrorCode::ZeroAmountSpecified);
```

A more general edge case may arise during a swap:

1. compute_swap first consumes most of the user-specified amount against active liquidity and moves execution to next_initialized_tick, leaving only a small state.amount_specified_remaining
2. match_limit_order is then invoked at that boundary
3. At some prices, the remaining result.amount_in may be too small to produce even one unit of output after rounding in get_limit_order_output, causing result.amount_out to become 0

4. The check at `programs/amm/src/states/tick_array.rs:542` then reverts the entire swap with ZeroAmountSpecified

Any sufficiently small remaining exact-input amount may trigger the same behavior when the limit-order output rounds down to zero.

Resolution

Resolved by commit 27c9252.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[L-05] Legacy swap fails when pool price remains unchanged

Identified in commit `c678919`.

Both `swap` and `swap_v2` share the same `swap_internal` logic. After the new limit order feature is added, `swap_internal` might only consume the limit orders placed on a tick, and the pool price will not move. This violates the requirement in the older version of the code, which specifies that the price must change towards the swap direction after a swap is completed.

The `exact_internal_v2` corresponding to the new `swap_v2` has modified this requirement to allow the price to remain unchanged. However, the `exact_internal` corresponding to `swap` still requires that the price must change before and after the swap. This will prevent users from completing their swaps normally when the situation described above occurs.

```
/* programs/amm/src/instructions/swap_v2.rs */
076 | pub fn exact_internal_v2<'c: 'info, 'info>(
082 | ) -> Result<u64> {
306 |     if zero_for_one {
307 |         require_gte!(swap_price_before, pool_state.sqrt_price_x64);
308 |     } else {
309 |         require_gte!(pool_state.sqrt_price_x64, swap_price_before);
310 |     }
339 | }

/* programs/amm/src/instructions/swap.rs */
772 | pub fn exact_internal<'b, 'c: 'info, 'info>(
778 | ) -> Result<u64> {
928 |     if zero_for_one {
929 |         require_gt!(swap_price_before, pool_state.sqrt_price_x64);
930 |     } else {
931 |         require_gt!(pool_state.sqrt_price_x64, swap_price_before);
932 |     }
961 | }
```

This situation only happens when the pool price lands exactly on the tick price. This sounds unlikely to occur, but it is entirely possible if the order volume placed on a tick is large. A certain swap might not be able to completely consume the orders on the tick, causing the pool price to stop exactly at the tick price. Subsequent users who use the `swap` interface instead of the `swap_v2` interface might encounter the DoS issue mentioned above.

We recommend changing the requirement in `exact_internal` to also allow for equality.

Resolution

Resolved by commit 5ec3081.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[I-01] `settle_filled_order` output loses precision on repeated partial settlements

Identified in commit `c678919`.

The `settle_filled_order` can be called multiple times on the same order (each call to `settle_limit_order` or `decrease_amount` triggers it). Each invocation computes `filled_amount` as the difference between `remaining_amount` and a ceiling-rounded `new_remaining_amount`, then converts it to output tokens via `get_limit_order_output` which uses floor division:

```
/* programs/amm/src/states/tick_array.rs */
404 | /// Get the output amount of a limit order
405 | /// amount_in: the amount of the input token
406 | /// zero_for_one: the direction of the input token
407 | /// output amount is rounded down
408 | pub fn get_limit_order_output(amount_in: u64, tick: i32, zero_for_one: bool)
```

Because `get_limit_order_output` rounds down and each settlement converts only the incremental `filled_amount`, splitting a settlement into multiple calls may yield a slightly smaller total output than settling in one call. The rounding error per call is at most 1 base unit, but it accumulates over repeated settlements. This is favorable to the protocol (users receive slightly less), but may cause minor discrepancies between expected and actual output amounts.

Resolution

The team acknowledged this finding.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[I-02] Base fee rate not capped at `MAX_FEE_RATE_NUMERATOR` in static-fee pools

Identified in commit `c678919`.

The `get_total_fee_rate` function applies the `MAX_FEE_RATE_NUMERATOR` cap (100,000 = 10%) only when dynamic fees are enabled. For static-fee pools, the base fee rate is returned without any cap:

```
/* programs/amm/src/instructions/swap.rs */
400 | pub fn get_total_fee_rate(&self) -> Result<u32> {
401 |     // Use base + dynamic fee if dynamic fee is enabled
402 |     if let Some(dynamic_fee_info) = &self.dynamic_fee_info {
403 |         let dynamic_fee_rate =
404 |             Self::compute_dynamic_fee_rate(dynamic_fee_info, self.tick_spacing)?;
405 |         let total_fee_rate = self.base_fee_rate + dynamic_fee_rate;
406 |         return Ok(total_fee_rate.min(MAX_FEE_RATE_NUMERATOR));
407 |     }
408 |     // Use base fee if not in launch phase and dynamic fee is disabled
409 |     Ok(self.base_fee_rate)
410 | }
```

The `update_trade_fee_rate` admin function allows setting values up to 999,999 (99.9999%):

```
/* programs/amm/src/states/config.rs */
006 | pub const FEE_RATE_DENOMINATOR_VALUE: u32 = 1_000_000;
/* programs/amm/src/instructions/admin/update_amm_config.rs */
060 | fn update_trade_fee_rate(amm_config: &mut Account<AmmConfig>, trade_fee_rate: u32) -> Result<()> {
061 |     require_gt!(FEE_RATE_DENOMINATOR_VALUE, trade_fee_rate);
062 |     amm_config.trade_fee_rate = trade_fee_rate;
063 |     Ok(())
064 | }
```

This creates an inconsistency: dynamic-fee pools are capped at a 10% total fee, but static-fee pools can theoretically charge up to 99.9999%.

Resolution

The team acknowledged this finding.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[I-03] `DynamicFeeConfig` validated with hardcoded `tick_spacing`

Identified in commit `c678919`.

When a `DynamicFeeConfig` is created or updated, the overflow check in `validate_params` uses a hardcoded `tick_spacing` of `1000`:

```
/* programs/amm/src/states/dynamic_fee_config.rs */
044 | pub fn update_dynamic_fee_config(
051 | ) -> Result<()> {
052 |     if !DynamicFeeInfo::validate_params(
053 |         1000,
054 |         filter_period,
055 |         decay_period,
056 |         reduction_factor,
057 |         dynamic_fee_control,
058 |         max_volatility_accumulator,
059 |     ) {
060 |         return Err(ErrorCode::InvalidDynamicFeeConfigParams.into());
061 |     }

/* programs/amm/src/states/pool_fee.rs */
093 | // To ensure safety in dynamic fee calculation, max_volatility_accumulator * tick_spacing must not cause an
↪  integer overflow
094 | if u64::from(max_volatility_accumulator) * u64::from(tick_spacing) > u32::MAX as u64 {
095 |     return false;
096 | }
```

However, when the config is applied to a pool via `initialize_dynamic_fee_info`, the same `validate_params` check is re-run with the pool's actual `tick_spacing`:

```
/* programs/amm/src/states/pool_fee.rs */
186 | impl PoolState {
187 |     pub fn initialize_dynamic_fee_info(
195 |     ) -> Result<()> {
196 |         if !DynamicFeeInfo::validate_params(
197 |             self.tick_spacing,
198 |             filter_period,
199 |             decay_period,
200 |             reduction_factor,
201 |             dynamic_fee_control,
202 |             max_volatility_accumulator,
203 |         ) {
204 |             return Err(ErrorCode::InvalidDynamicFeeConfigParams.into());
205 |         }
```

This creates a mismatch: configs that would be safe for pools with `tick_spacing <= 1000` may be unnecessarily rejected at config creation time (e.g., if `max_volatility_accumulator * 1000 > u32::MAX` but `max_volatility_accumulator * actual_tick_spacing` would be fine). Conversely, configs that pass val-

idation at `tick_spacing=1000` might not pass at pool creation time if the pool's actual `tick_spacing > 1000`. Since `initialize_dynamic_fee_info` re-validates with the real `tick_spacing`. valid configs may be rejected prematurely, and invalid configs will be caught at pool creation.

Resolution

The team acknowledged this finding.

FEAT_LIMITORDER_DYNAMICFEE DIFF

[I-04] Decrease rebasing can discard deferred settlement dust

Identified in commit `f1ec7dc`.

The limit-order settlement path is meant to pay partially matched orders incrementally while recovering any dust that was held back earlier once the segment becomes fully filled. In the partial-fill branch, however, `settle_filled_order` reduces the economically payable fill by one unit whenever the ratio is non-exact, but still advances `filled_amount` using `ideal_remaining`. The same code also relies on a later full-fill branch to recover that deferred amount.

```
/* programs/amm/src/states/limit_order.rs */
118 | let effective_filled = if is_exact {
119 |     total_filled
120 | } else {
121 |     total_filled.saturating_sub(1)
122 | };
123 |
124 | let total_output = TickState::get_limit_order_output(
125 |     effective_filled,
126 |     tick_state.tick,
127 |     self.zero_for_one,
128 | )?;
129 |
130 | let payout = total_output.saturating_sub(self.settled_output);
131 |
132 | // Always update filled_amount even if payout is 0 (e.g. filled=1,
133 | // effective=0   output=0), so get_unfilled_amount() stays correct.
134 | self.filled_amount = self
135 |     .total_amount
136 |     .checked_sub(ideal_remaining)
137 |     .ok_or(ErrorCode::CalculateOverflow)?;
138 | self.settled_output = total_output;

141 | } else if self.order_phase + 2 <= tick_state.order_phase {
142 |     // Fully filled: all remaining consumed. Output diff recovers any dust
143 |     // held back during partial-fill settles.
144 |     let total_output = TickState::get_limit_order_output(
145 |         self.settle_base,
146 |         tick_state.tick,
147 |         self.zero_for_one,
148 |     )?;
149 |     let payout = total_output.saturating_sub(self.settled_output);
150 |
151 |     self.filled_amount = self.total_amount;
152 |     self.settle_base = 0;
153 |     self.settled_output = 0;

223 | // Reset computation baseline for the new segment
224 | if self.order_phase + 1 == tick_state.order_phase {
225 |     let new_unfilled = self.get_unfilled_amount()?;
226 |     self.settle_base = new_unfilled;
227 |     self.unfilled_ratio_x64 = tick_state.unfilled_ratio_x64;
```

```
228 |     self.settled_output = 0;
229 | } else {
```

The output conversion itself is rounded down, so the withheld unit remains as deferred dust rather than an immediately claimable payout.

```
/* programs/amm/src/states/tick_array.rs */
401 | /// Get the output amount of a limit order
402 | /// amount_in: the amount of the input token
403 | /// zero_for_one: the direction of the input token
404 | /// output amount is rounded down
405 | pub fn get_limit_order_output(amount_in: u64, tick: i32, zero_for_one: bool) -> Result<u64> {
406 |     let output_amount = if zero_for_one {
407 |         let token_0_price_x64 = tick_math::get_price_at_tick(tick, false)?;
408 |         // Convert token0 amount to token1 amount using token0 price
409 |         // token1_amount = token0_amount * token_0_price_x64 / 2^64
410 |         U128::from(amount_in)
411 |             .mul_div_floor(token_0_price_x64, U128::from(fixed_point_64::Q64))
412 |             .ok_or(ErrorCode::CalculateOverflow)?
413 |             .as_u64()
414 |     } else {
415 |         let token_0_price_x64 = tick_math::get_price_at_tick(tick, true)?;
416 |         // Convert token1 amount to token0 amount using token1 price (1/token_0_price_x64)
417 |         // token0_amount = token1_amount * 2^64 / token_0_price_x64
418 |         U128::from(amount_in)
419 |             .mul_div_floor(U128::from(fixed_point_64::Q64), token_0_price_x64)
420 |             .ok_or(ErrorCode::CalculateOverflow)?
421 |             .as_u64()
422 |     };
423 |     Ok(output_amount)
```

If a user calls `decrease_amount` after such a non-exact partial match, the new segment is rebased from `get_unfilled_amount()`, which has already been reduced by the unpaid unit recorded in `filled_amount`. Resetting `settle_base` and `settled_output` at that point drops the deferred dust before the later full-fill branch can recover it.

The test `test_repeated_decrease_amount_loss` shows that repeating this sequence produces a persistent shortfall relative to the ideal output, with `loss >= decrease_times` in the reproduced `tick = 0` configuration.

This appears to be a low-severity edge case because it requires repeated interleaving of non-exact partial matches and decreases, and the loss remains bounded, but it may still cause users to receive less output than the final order size would otherwise imply.

The resulting behavior is limited to bounded dust loss, which we consider acceptable.

Test

```
#[test]
fn test_repeated_decrease_amount_loss() {
    let tick = 0;
    let initial_order_phase = 1_000_000;
    let limit_zero_for_one = true;
    let order_amount: u64 = 10_000;
    let match_chunk: u64 = 137; // guaranteed to break exactness
    let decrease_times: u64 = 10;

    let mut tick_state = create_mock_tick_state(tick, initial_order_phase, 0, 0);
    let mut order = open_order(
        order_amount,
        limit_zero_for_one,
        initial_order_phase,
        &mut tick_state,
    );

    let mut total_received = 0u64;

    // matches a chunk
    tick_state
        .match_limit_order(match_chunk, !limit_zero_for_one, false, 0, true)
        .unwrap();
    total_received += order.settle_filled_order(&tick_state).unwrap();

    //   1. decreases by 1
    //   2. matches a chunk
    //   3. settles
    for _ in 0..decrease_times {
        let r = order.decrease_amount(&mut tick_state, 1).unwrap();
        total_received += r.settled_output_amount;

        tick_state
            .match_limit_order(match_chunk, !limit_zero_for_one, false, 0, true)
            .unwrap();
        total_received += order.settle_filled_order(&tick_state).unwrap();
    }

    let remaining_in_tick = tick_state.part_filled_orders_remaining;
    if remaining_in_tick > 0 {
        tick_state
            .match_limit_order(remaining_in_tick, !limit_zero_for_one, false, 0, true)
            .unwrap();
    }
    total_received += order.settle_filled_order(&tick_state).unwrap();
    assert!(order.is_fully_filled(), "order should be fully filled");

    assert_eq!(order.total_amount, 10_000 - decrease_times);
    let ideal_output =
        TickState::get_limit_order_output(order.total_amount, tick, limit_zero_for_one)
            .unwrap();
    let loss = ideal_output - total_received;

    println!(
        "decrease_times = {}, ideal_output = {}, \
         total_received = {}, loss = {}",
```

```
        decrease_times, ideal_output, total_received, loss
    );

    assert!(
        loss >= decrease_times,
        "expected loss >= decrease_times ({}), got {} (total_received={}, ideal={})",
        decrease_times,
        loss,
        total_received,
        ideal_output
    );
}
```

Resolution

The team acknowledged this finding and clarified that this approach represents the most practical solution they identified to minimize calculation error.

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.

