Security Assessment Report

Orca Whirlpools

PRs 918, 902, 903, 904 and 970

June 23, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the PRs 918, 902, 903, 904 and 970 of the Orca Whirlpools program.

The artifact of the audit was the source code of the following programs, excluding tests, in https://github.com/orca-so/whirlpools.

The initial audit focused on the following versions and revealed 13 issues or questions.

#	pr	type	commit
P1	PR#918: Adaptive Fee Release	solana	8565ac6, a1772ad, b786159
P2	PR#902: Reset position range	solana	c2ea428
P3	PR#903: Allow locking concentrated positions	solana	dcc8379
P4	PR#904: Transfer locked position	solana	7bd6380
P5	PR#970: Dynamic TickArray	solana	318dc9f

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

Table of Contents

Result Overview

Issue	Impact	Status
PR#918: ADAPTIVE FEE RELEASE		
[P1-M-01] Malicious user can keep high adaptive fee at low cost	Medium	Resolved
[P1-L-01] Not skipping liquidity gaps when fetching the last tick group index	Low	Resolved
[P1-L-02] Inconsistent volatility accumulator updates	Low	Resolved
[P1-I-01] Inaccurate tick group index update conditions	Info	Resolved
[P1-Q-01] Question on the `adaptive_fee_rate` cap	Question	Resolved
[P1-Q-02] The `tick_group_index_reference` is not updated in edge cases	Question	Resolved
[P1-Q-03] Inconsistent comments	Question	Resolved
PR#902: RESET POSITION RANGE		
[P2-I-01] Unresolved comment	Info	Resolved
[P2-Q-01] Potential redundant checkpoint reset	Question	Resolved
PR#903: ALLOW LOCKING CONCENTRATED POSITIONS		
No issues found		
PR#904: TRANSFER LOCKED POSITION		
No issues found		
PR#970: DYNAMIC TICKARRAY		
[P5-I-01] Redundant condition	Info	Resolved
[P5-I-02] Unused function	Info	Resolved
[P5-Q-01] Potential incorrect error code and comment	Question	Resolved
[P5-Q-02] Will only variable-sized tick arrays be supported in the future?	Question	Resolved

Findings in Detail

PR#918: ADAPTIVE FEE RELEASE

[P1-M-01] Malicious user can keep high adaptive fee at low cost

Identified in commit `8565ac6`.

For a `Whirlpool` that enables the `AdaptiveFee` feature, the pool's oracle will track the last swap timestamp to determine if the pool is in the high-frequency trading mode. If yes, the final adaptive fee ratio will be higher.

```
/* programs/whirlpool/src/state/oracle.rs */
089 | pub struct AdaptiveFeeVariables {
090 |     // Last timestamp (block time) the variables was updated
091 |     pub last_update_timestamp: u64,
```

In the `update_reference` function, the `last_swap_timestamp` is updated with each swap, and the swap only checks that the input token amount is not zero.

```
/* programs/whirlpool/src/state/oracle.rs */
120 | pub fn update_reference(
121 |     &mut self,
122 |     tick_group_index: i32,
123 |     current_timestamp: u64,
124 |     adaptive_fee_constants: &AdaptiveFeeConstants,
125 | ) -> Result<()> {
126 |     if current_timestamp < self.last_update_timestamp {
127 |         return Err(ErrorCode::InvalidTimestamp.into());
128 |     }
129 |
130 |     let elapsed = current_timestamp - self.last_update_timestamp;
131 |
132 |     if elapsed < adaptive_fee_constants.filter_period as u64 {
133 |         // high frequency trade
134 |         // no change
135 |     } else if elapsed < adaptive_fee_constants.decay_period as u64 {
136 |         // NOT high frequency trade
137 |         self.tick_group_index_reference = tick_group_index;
138 |         self.volatility_reference = (u64::from(self.volatility_accumulator)
139 |             * u64::from(adaptive_fee_constants.reduction_factor)
140 |             / u64::from(REDUCTION_FACTOR_DENOMINATOR))
141 |             as u32;
142 |     } else {
143 |         // Out of decay time window
```

```
144 |         self.tick_group_index_reference = tick_group_index;
145 |         self.volatility_reference = 0;
146 |     }
147 |
148 |     self.last_update_timestamp = current_timestamp;
```

```
/* programs/whirlpool/src/manager/swap_manager.rs */
060 | if amount == 0 {
061 |     return Err(ErrorCode::ZeroTradableAmount.into());
062 | }
```

As a result, a malicious user could periodically trigger updates to `last_swap_timestamp` at a low cost (e.g., by swapping 1 smallest unit of a token). This would keep the `Whirlpool` in high-frequency trading mode, requiring a higher adaptive fee for swaps.

This issue has been acknowledged in Trader Joe (Code4rena report). Trader Joe addressed this by introducing a `forceDecay` function, enabling the admin to reset `index_reference` and `volatility_reference` when necessary.

An alternative workaround is to update `last_swap_timestamp` only when the swap amount is significant, such as when it crosses a tick group. This prevents minor, negligible swaps from artificially maintaining the fee at its peak.

Resolution

The team resolved by the commit orca-so/whirlpools @ 39b3c74, which introduced the `major_swap` checks.

PR#918: ADAPTIVE FEE RELEASE

[P1-L-01] Not skipping liquidity gaps when fetching the last tick group index

Identified in commit `8565ac6`.

In the swap function, the swap process crosses multiple tick groups between the `current tick_index` and `next_initialized_tick_index` within a loop. The minimal swap step is between `curr_sqrt_price` and `bounded_sqrt_price_target`.

```
/* programs/whirlpool/src/manager/swap_manager.rs */
106 | let bounded_sqrt_price_target =
107 |     fee_rate_manager.get_bounded_sqrt_price_target(sqrt_price_target);
108 |
109 | let swap_computation = compute_swap(
110 |     amount_remaining,
111 |     total_fee_rate,
112 |     curr_liquidity,
113 |     curr_sqrt_price,
114 |     bounded_sqrt_price_target,
115 |     amount_specified_is_input,
116 |     a_to_b,
117 | )?;
```

When crossing a tick group (`curr_sqrt_price == bounded_sqrt_price_target`), the `fee_rate_manager` is updated accordingly.

```
/* programs/whirlpool/src/manager/swap_manager.rs */
211 | curr_sqrt_price = swap_computation.next_price;
213 | if curr_sqrt_price == bounded_sqrt_price_target {
214 |     fee_rate_manager.advance_tick_group();
215 | }
```

If `sqrt_price_target` is reached, the loop terminates and retrieves the new `next_initialized_tick_index`. The swap proceeds within the new liquidity range.

```
/* programs/whirlpool/src/manager/swap_manager.rs */
218 | if amount_remaining == 0 || curr_sqrt_price == sqrt_price_target {
219 |     break;
220 | }

/* programs/whirlpool/src/manager/swap_manager.rs */
091 | let (next_array_index, next_tick_index) = swap_tick_sequence
092 |             .get_next_initialized_tick_index(
```

```
093 |                 curr_tick_index,
094 |                 tick_spacing,
095 |                 a_to_b,
096 |                 curr_array_index,
097 |             )?;
098 |
099 |         let (next_tick_sqrt_price, sqrt_price_target) =
100 |             get_next_sqrt_prices(next_tick_index, adjusted_sqrt_price_limit, a_to_b);
```

However, the `advance_tick_group` function only increments or decrements the tick group index by `1` without verifying liquidity presence in the new tick group.

For liquidity gaps where the last tick group index is far from the previous one, swaps still traverse each tick group sequentially. As a result, this renders the `get_next_initialized_tick_index` function ineffective.

```
/* programs/whirlpool/src/manager/fee_rate_manager.rs */
090 | pub fn advance_tick_group(&mut self) {
091 |     match self {
092 |         Self::Static { .. } => {
093 |             // do nothing
094 |         }
095 |         Self::Adaptive {
096 |             a_to_b,
097 |             tick_group_index,
098 |             ..
099 |         } => {
100 |             *tick_group_index += if *a_to_b { -1 } else { 1 };
101 |         }
102 |     }
103 | }
```

Consider the following scenario. Assume

- Current `tick_index` is `0`.
- Tick group size is `2` so the current tick group index is `0 % 2 = 0`.
- Tick group reference is `0`.
- Liquidity exists in the tick ranges `[0, 2]` and `[8, 10]`.
- Swap executed across `[0,10]`.

During the swap, for the first liquidity range, the `tick_index` changes from `0` to `2`. The current tick group index is updated from `0` to `1`.

The `next_initialized_tick_index` becomes `8`. The swap should proceed in range `[8, 10]` while skipping `[2, 8]`.

However, the swap loop still incrementally processes tick groups:

- Group 1 → 2 (range `[2, 4]`)
- Group 2 → 3 (range `[4, 6]`)
- Group 3 → 4 (range `[6, 8]`)
- ...

This forces inefficient step-by-step execution despite available liquidity gaps.

Consider modifying the `advance_tick_group` to skip liquidity-empty tick groups.

Resolution

The issue was fixed by orca-so/whirlpools @ 134c8b2, which introduced the `advance_tick_group_after_skip` function.

PR#918: ADAPTIVE FEE RELEASE

[P1-L-02] Inconsistent volatility accumulator updates

Identified in commit `a1772ad`.

In the `adaptive_fee_update_skipped` mode, the `advance_tick_group_after_skip` function always updates the `volatility_accumulator` using the newest `tick_group_index` (**including at the end of the swap**).

```
/* programs/whirlpool/src/manager/fee_rate_manager.rs */
158 | pub fn advance_tick_group_after_skip(
159 |     &mut self,
160 |     sqrt_price: u128,
161 |     next_tick_sqrt_price: u128,
162 |     next_tick_index: i32,
163 | ) -> Result<()> {
164 |     match self {
165 |         Self::Static { .. } => {
166 |             // static fee rate manager doesn't use skip feature
167 |             unreachable!();
168 |         }
169 |         Self::Adaptive {
170 |             a_to_b,
171 |             tick_group_index,
172 |             adaptive_fee_variables,
173 |             adaptive_fee_constants,
175 |         } => {
176 |             if sqrt_price == next_tick_sqrt_price {
177 |                 // next_tick_index = tick_index_from_sqrt_price(&sqrt_price) is true,
178 |                 // but we use next_tick_index to reduce calculations in the middle of the loop
179 |                 *tick_group_index = floor_division(
180 |                     next_tick_index,
181 |                     adaptive_fee_constants.tick_group_size as i32,
182 |                 );
183 |             } else {
184 |                 // End of the swap loop or the boundary of core tick group range.
186 |                 // Note: It was pointed out during the review that using curr_tick_index may
↪  suppress tick_index_from_sqrt_price.
187 |                 //       However, since curr_tick_index may also be shifted by -1, we decided to
↪  prioritize safety by recalculating it here.
188 |                 *tick_group_index = floor_division(
189 |                     tick_index_from_sqrt_price(&sqrt_price),
190 |                     adaptive_fee_constants.tick_group_size as i32,
191 |                 );
192 |             }
194 |             // volatility_accumulator is updated with the new tick_group_index based on new
↪  sqrt_price
195 |             adaptive_fee_variables
196 |                 .update_volatility_accumulator(*tick_group_index, adaptive_fee_constants)?;
```

However, when `! adaptive_fee_update_skipped` is `true`, the `volatility_accumulator` will not be updated with the newest `tick_group_index`, even though the last swap step crosses a tick group.

In particular, `curr_sqrt_price != bounded_sqrt_price_target` indicates the end of the swap. The last swap step (swap end) may occur when `curr_sqrt_price == bounded_sqrt_price_target`.

For example, when `adjusted_sqrt_price_limit == sqrt_price_target == bounded_sqrt_price_target == curr_sqrt_price`, the remaining swap amount is not zero, while the `curr_sqrt_price` has already reached the `adjusted_sqrt_price_limit`. At this moment, the `tick_group_index` will be updated to the new tick group, and the `volatility_accumulator` will not be updated.

```
/* programs/whirlpool/src/manager/swap_manager.rs */
213 | if !adaptive_fee_update_skipped {
214 |     // Note: curr_sqrt_price != bounded_sqrt_price_target implies the end of the loop.
215 |     //       tick_group_index counter exists only in the memory of the FeeRateManager,
216 |     //       so even if it is incremented one extra time at the end of the loop, there is no real
↪  harm.
217 |     fee_rate_manager.advance_tick_group();
218 | } else {

/* programs/whirlpool/src/manager/fee_rate_manager.rs */
141 | // This function is called when skip is NOT used.
142 | pub fn advance_tick_group(&mut self) {
143 |     match self {
144 |         Self::Static { .. } => {
145 |             // do nothing
146 |         }
147 |         Self::Adaptive {
148 |             a_to_b,
149 |             tick_group_index,
150 |             ..
151 |         } => {
152 |             *tick_group_index += if *a_to_b { -1 } else { 1 };
153 |         }
154 |     }
155 | }

/* programs/whirlpool/src/manager/swap_manager.rs */
226 | // do while loop
227 | if amount_remaining == 0 || curr_sqrt_price == sqrt_price_target {
228 |     break;
229 | }

/* programs/whirlpool/src/manager/swap_manager.rs */
090 | while amount_remaining > 0 && adjusted_sqrt_price_limit != curr_sqrt_price {
```

Consider the following scenario in the `adaptive_fee_update_skipped` mode.

- The current `tick_index` is `0`.
- The tick group size is `2`.
- The current `tick_group_index` is `0 % 2 = 0`.
- Liquidity exists in the tick range `[0, 6)`, which covers:
  - Tick group 0: `[0, 2)`
  - Tick group 1: `[2, 4)`
  - Tick group 2: `[4, 6)`
- The swap amount is large enough to potentially cross the entire tick range `[0, 6)`. However, the `adjusted_sqrt_price_limit` is set at tick `2`.

The `volatility_accumulator` updates:

- Loop 1: `update_volatility_accumulator` is called with tick group 0. The current `tick_index` moves from 0 to 2, the actual swap occurs within tick group 0, and the swap ends.
- Therefore, tick index 2 should not be treated as tick group 1 for the updating `volatility_accumulator`.

It is recommended to update the `volatility_accumulator` consistently between `! adaptive_fee_update_skipped` and `adaptive_fee_update_skipped` mode.

Resolution

The `advance_tick_group_after_skip` function was updated to keep volatility accumulator updates consistent in the commit orca-so/whirlpools @ ab64b1a.

PR#918: ADAPTIVE FEE RELEASE

[P1-I-01] Inaccurate tick group index update conditions

Identified in commit `10fa8b7`.

The `swap` function invokes the `advance_tick_group()` and updates the `tick_group_index` if it crosses a tick group.

```
/* whirlpools-staging/programs/whirlpool/src/manager/swap_manager.rs */
030 | pub fn swap(
039 | ) -> Result<PostSwapUpdate> {
090 |     while amount_remaining > 0 && adjusted_sqrt_price_limit != curr_sqrt_price {
102 |         loop {
105 |             let total_fee_rate = fee_rate_manager.get_total_fee_rate();
106 |             let (bounded_sqrt_price_target, adaptive_fee_update_skipped) =
107 |                 fee_rate_manager.get_bounded_sqrt_price_target(sqrt_price_target, curr_liquidity);
213 |             if !adaptive_fee_update_skipped {
214 |                 fee_rate_manager.advance_tick_group();
215 |             } else {
216 |                 fee_rate_manager.advance_tick_group_after_skip(
217 |                     curr_sqrt_price,
218 |                     next_tick_sqrt_price,
219 |                     next_tick_index,
220 |                 )?;
221 |             }

/* programs/whirlpool/src/manager/fee_rate_manager.rs */
141 | // This function is called when skip is NOT used.
142 | pub fn advance_tick_group(&mut self) {
143 |     match self {
144 |         Self::Static { .. } => {
145 |             // do nothing
146 |         }
147 |         Self::Adaptive {
148 |             a_to_b,
149 |             tick_group_index,
150 |             ..
151 |         } => {
152 |             *tick_group_index += if *a_to_b { -1 } else { 1 };
153 |         }
154 |     }
155 | }
```

However, the condition for invoking the `advance_tick_group` function is inaccurate, which only requires that the flag `adaptive_fee_update_skipped` is false, without checking whether the `curr_sqrt_price` equals the `bounded_sqrt_price_target` (swap cross a tick group).

As a result, the `tick_group_id` is always updated, even if the swap does not cross a tick group.

It is recommended to modify the condition from `if !adaptive_fee_update_skipped` to `if !adaptive_fee_update_skipped && curr_sqrt_price == bounded_sqrt_price_target`.

Resolution

The team added a comment in orca-so/whirlpools @ a1772ad and clarified that there is no real harm even if it is incremented one extra time at the end of the loop.

PR#918: ADAPTIVE FEE RELEASE

[P1-Q-01] Question on the `adaptive_fee_rate` cap

Identified in commit `3e79260`.

According to the protocol design, the `total_fee_rate` is defined as the sum of `adaptive_fee_rate` and `static_fee_rate`.

```
/* programs/whirlpool/src/manager/fee_rate_manager.rs */
105 | pub fn get_total_fee_rate(&self) -> u32 {
106 |     match self {
107 |         Self::Static { static_fee_rate } => *static_fee_rate as u32,
108 |         Self::Adaptive {
109 |             static_fee_rate,
110 |             adaptive_fee_constants,
111 |             adaptive_fee_variables,
112 |             ..
113 |         } => {
114 |             let adaptive_fee_rate =
115 |                 Self::compute_adaptive_fee_rate(adaptive_fee_constants, adaptive_fee_variables);
116 |             let total_fee_rate = *static_fee_rate as u32 + adaptive_fee_rate;
117 |
```

The `compute_adaptive_fee_rate` function calculates the `adaptive_fee_rate` and includes a check to ensure it does not exceed `FEE_RATE_HARD_LIMIT`.

```
/* programs/whirlpool/src/manager/fee_rate_manager.rs */
186 | if fee_rate > FEE_RATE_HARD_LIMIT as u128 {
187 |     FEE_RATE_HARD_LIMIT
188 | } else {
189 |     fee_rate as u32
190 | }
```

However, since the `total_fee_rate` is capped at `FEE_RATE_HARD_LIMIT`, should the `adaptive_fee_rate` be capped at `FEE_RATE_HARD_LIMIT - static_fee_rate` to prevent `total_fee_rate` from exceeding this limit?

```
/* programs/whirlpool/src/manager/fee_rate_manager.rs */
118 | if total_fee_rate > FEE_RATE_HARD_LIMIT {
119 |     FEE_RATE_HARD_LIMIT
120 | } else {
121 |     total_fee_rate
122 | }
```

Resolution

The team clarified that this is the desired behavior: Whether the `compute_adaptive_fee_rate` should be capped using `FEE_RATE_HARD_LIMIT` or `FEE_RATE_HARD_LIMIT - static_fee_rate` depends on the intended role of `compute_adaptive_fee_rate`:

- **Option 1**: Cap based on the maximum possible total fee rate, taking `static_fee_rate` into account
- **Option 2**: Treat `adaptive_fee_rate` as independent from `static_fee_rate`, and apply a global cap to the final combined fee rate

Both options should work.

The key point is that the adaptive fee rate is calculated independently of the static fee rate, but is ultimately subject to capping. So, it's a design choice the team made that the adaptive fee rate alone can reach the `FEE_RATE_HARD_LIMIT`.

PR#918: ADAPTIVE FEE RELEASE

[P1-Q-02] The `tick_group_index_reference` is not updated in edge cases

Identified in commit `8565ac6`.

According to the protocol design, the adaptive fee is calculated based on changes in the tick group index (`index_delta`) during the swap process.

Before the swap begins, the function `FeeRateManager::new()` initializes the `tick_group_index_reference` value by calling `adaptive_fee_variables.update_reference`.

```
/* programs/whirlpool/src/manager/swap_manager.rs */
082 | let mut fee_rate_manager = FeeRateManager::new(
083 |     a_to_b,
084 |     whirlpool.tick_current_index, // note:  -1 shift is acceptable
085 |     timestamp,
086 |     fee_rate,
087 |     adaptive_fee_info,
088 | )?;

/* programs/whirlpool/src/manager/fee_rate_manager.rs */
042 | pub fn new(
043 |     a_to_b: bool,
044 |     current_tick_index: i32,
045 |     timestamp: u64,
046 |     static_fee_rate: u16,
047 |     adaptive_fee_info: Option<AdaptiveFeeInfo>,
048 | ) -> Result<Self> {
049 |     match adaptive_fee_info {
050 |         None => Ok(Self::Static { static_fee_rate }),
051 |         Some(adaptive_fee_info) => {
052 |             let tick_group_index = floor_division(
053 |                 current_tick_index,
054 |                 adaptive_fee_info.constants.tick_group_size as i32,
055 |             );
056 |             let adaptive_fee_constants = adaptive_fee_info.constants;
057 |             let mut adaptive_fee_variables = adaptive_fee_info.variables;
059 |             // update reference at the initialization of the fee rate manager
060 |             adaptive_fee_variables.update_reference(
061 |                 tick_group_index,
062 |                 timestamp,
063 |                 &adaptive_fee_constants,
064 |             )?;
```

Subsequently, during the swap process, the function `update_volatility_accumulator` calculates the `index_delta` using `tick_group_index_reference - tick_group_index`.

```
/* programs/whirlpool/src/manager/swap_manager.rs */
102 | loop {
103 |     fee_rate_manager.update_volatility_accumulator()?;
105 |     let total_fee_rate = fee_rate_manager.get_total_fee_rate();

/* programs/whirlpool/src/state/oracle.rs */
103 | pub fn update_volatility_accumulator(
104 |     &mut self,
105 |     tick_group_index: i32,
106 |     adaptive_fee_constants: &AdaptiveFeeConstants,
107 | ) -> Result<()> {
108 |     let index_delta = (self.tick_group_index_reference - tick_group_index).unsigned_abs();
```

In the `update_reference` function, if the time elapsed since the last swap is less than the `filter_period`, the `tick_group_index_reference` will not be updated.

However, after each swap concludes, the `tick_group_index_reference` is not updated to the current tick group index.

```
/* programs/whirlpool/src/state/oracle.rs */
120 | pub fn update_reference(
121 |         &mut self,
122 |         tick_group_index: i32,
123 |         current_timestamp: u64,
124 |         adaptive_fee_constants: &AdaptiveFeeConstants,
125 |     ) -> Result<()> {
126 |         if current_timestamp < self.last_update_timestamp {
127 |             return Err(ErrorCode::InvalidTimestamp.into());
128 |         }
130 |         let elapsed = current_timestamp - self.last_update_timestamp;
132 |         if elapsed < adaptive_fee_constants.filter_period as u64 {
133 |             // high frequency trade
134 |             // no change
135 |         } else if elapsed < adaptive_fee_constants.decay_period as u64 {
136 |             // NOT high frequency trade
137 |             self.tick_group_index_reference = tick_group_index;
138 |             self.volatility_reference = (u64::from(self.volatility_accumulator)
139 |                 * u64::from(adaptive_fee_constants.reduction_factor)
140 |                 / u64::from(REDUCTION_FACTOR_DENOMINATOR))
141 |                 as u32;
142 |         } else {
143 |             // Out of decay time window
144 |             self.tick_group_index_reference = tick_group_index;
145 |             self.volatility_reference = 0;
146 |         }
148 |         self.last_update_timestamp = current_timestamp;
150 |         Ok(())
151 |     }
152 | }
```

This results in the adaptive fee for swaps occurring within the `filter_period` being calculated based on an inaccurate `index_delta`.

Consider the following scenario. Assume the `current tick index = 0`, tick group size is `10`, tick group index is `0 % 10 = 0`, and `filter_period` is 5 seconds.

1. Alice executes the first swap. The `tick_group_index_reference` is initialized to `0`. The swap updates the current tick index to `15`. The system calculates the `index_delta` as `(15 % 10) - 0 = 1`.
2. Three seconds later, Bob executes the second swap. With elapsed time still within the `filter_period`, the `tick_group_index_reference` retains `0`. Bob's swap changes the current tick index from `15` to `25`. The system calculates the `index_delta` as `(25 % 10) - 0 = 2`, though the correct value should be `(25 % 10) - 1 = 1`.

Consider updating the `tick_group_index_reference` even when elapsed time falls below the `filter_period`.

```
if elapsed < adaptive_fee_constants.filter_period as u64 {
    self.tick_group_index_reference = tick_group_index;
}
```

In fact, since the design is largely the same as the TraderJoe protocol at LBPair.sol#L515-L568, is this the intended behavior?

Resolution

The team clarified that not updating `tick_group_index_reference` when the `filter_period` is not reached is the intended behavior.

PR#918: ADAPTIVE FEE RELEASE

[P1-Q-03] Inconsistent comments

Identified in commit `10fa8b7`.

The comment in line 196 says the `tick_group_index` will advance by one more if `sqrt_price` is **not** on the `tick_group_size` boundary.

Is it supposed to be the opposite? i.e., the `tick_group_index` should advance by one more when the `sqrt_price` is on the `tick_group_size` boundary.

```
/* whirlpools-staging/programs/whirlpool/src/manager/fee_rate_manager.rs */
158 | pub fn advance_tick_group_after_skip(
159 |     &mut self,
160 |     sqrt_price: u128,
161 |     next_tick_sqrt_price: u128,
162 |     next_tick_index: i32,
163 | ) -> Result<()> {
164 |     match self {
169 |         Self::Adaptive {
170 |             a_to_b,
171 |             tick_group_index,
172 |             adaptive_fee_variables,
173 |             adaptive_fee_constants,
174 |             ..
175 |         } => {
176 |             if sqrt_price == next_tick_sqrt_price {
177 |                 // next_tick_index = tick_index_from_sqrt_price(&sqrt_price) is true,
178 |                 // but we use next_tick_index to reduce calculations in the middle of the loop
179 |                 *tick_group_index = floor_division(
180 |                     next_tick_index,
181 |                     adaptive_fee_constants.tick_group_size as i32,
182 |                 );
183 |             } else {
184 |                 // End of the swap loop
185 |                 *tick_group_index = floor_division(
186 |                     tick_index_from_sqrt_price(&sqrt_price),
187 |                     adaptive_fee_constants.tick_group_size as i32,
188 |                 );
189 |             }
195 |             // If the swap direction is A to B, the tick group index should be decremented to
↪  "advance".
196 |             // If sqrt_price is not on the tick_group_size boundary, tick_group_index will advance
↪  by one more.
197 |             // However, it does not affect subsequent processing because it is the last iteration
↪  of the swap loop.
198 |             if *a_to_b {
199 |                 *tick_group_index -= 1;
```

```
200 |             }
```

If so, when the `sqrt_price` is on the `tick_group_size` boundary, the corresponding `tick_group_index` should be subtracted by one to fetch the correct bound sqrt price. So, could the lines 198-200 be moved into the `if sqrt_price == next_tick_sqrt_price` branch?

Since the else branch at line 183 implies the end of the swap loop, it does not matter to leave the `*tick_group_index -= 1;` there though.

Resolution

The comment has been updated by the commit orca-so/whirlpools @ 570bfd6.

PR#902: RESET POSITION RANGE

[P2-I-01] Unresolved comment

Identified in commit `a8944e7`.

The `reset_position_range` function updates the position's lower and upper ticks to adjust the liquidity range.

The function checks the new lower and upper tick values to ensure they are valid. However, as implied by the comment "Do we care whether the tick range is the same as before?", it does not verify whether the new lower and upper ticks are the same as their previous values.

```
/* programs/whirlpool/src/state/position.rs */
094 | // Do we care whether the tick range is the same as before?
095 | validate_tick_range_for_whirlpool(whirlpool, new_tick_lower_index, new_tick_upper_index)?;
```

Since updating the lower and upper ticks to the same values seems redundant, adding a relevant check would be helpful.

Resolution

The team has added the relevant check in the commit orca-so/whirlpools @ b62cf69.

PR#902: RESET POSITION RANGE

[P2-Q-01] Potential redundant checkpoint reset

> *Identified in commit* `a8944e7`.

Function `reset_position_range` updates the position's lower/upper ticks and resets the `fee_growth_checkpoint_a` and `fee_growth_checkpoint_b`, which tracks the fee index.

```
/* programs/whirlpool/src/state/position.rs */
105 | // Reset the growth checkpoints
106 | self.fee_growth_checkpoint_a = 0;
107 | self.fee_growth_checkpoint_b = 0;
108 |
109 | // fee_owed and rewards.amount_owed should be zero due to the check above
110 | for i in 0..NUM_REWARDS {
111 |     self.reward_infos[i].growth_inside_checkpoint = 0;
112 | }
```

As mentioned in the comment "fee_owed and rewards.amount_owed should be zero due to the check above", the position's fees and rewards have already been withdrawn.

A position's fee and reward are calculated based on the delta between `fee_growth_inside_a` and `fee_growth_checkpoint_a`, multiplied by the liquidity amount. The latest fee growth index (`fee_growth_inside_a/b`) will be updated to the `fee_growth_checkpoint`.

```
/* programs/whirlpool/src/manager/position_manager.rs */
016 | // Calculate fee deltas.
017 |   // If fee deltas overflow, default to a zero value. This means the position loses
018 |   // all fees earned since the last time the position was modified or fees collected.
019 |   let growth_delta_a = fee_growth_inside_a.wrapping_sub(position.fee_growth_checkpoint_a);
020 |   let fee_delta_a = checked_mul_shift_right(position.liquidity, growth_delta_a).unwrap_or(0);
021 |
022 |   let growth_delta_b = fee_growth_inside_b.wrapping_sub(position.fee_growth_checkpoint_b);
023 |   let fee_delta_b = checked_mul_shift_right(position.liquidity, growth_delta_b).unwrap_or(0);
024 |
025 |   update.fee_growth_checkpoint_a = fee_growth_inside_a;
026 |   update.fee_growth_checkpoint_b = fee_growth_inside_b;
027 |
028 |   // Overflows allowed. Must collect fees owed before overflow.
029 |   update.fee_owed_a = position.fee_owed_a.wrapping_add(fee_delta_a);
030 |   update.fee_owed_b = position.fee_owed_b.wrapping_add(fee_delta_b);
```

As a result, the fees and rewards owed to a position are not directly affected by the absolute

values of `fee_growth_checkpoint_a` and `fee_growth_checkpoint_b`, but rather by their difference from the current `fee_growth_inside_a` and `fee_growth_inside_b`.

In other words, `fee_growth_checkpoint_a` and `fee_growth_checkpoint_b` could be initialized to arbitrary values without impacting correctness.

This is not a security concern. We are just curious whether resetting these values to zero is necessary.

Resolution

The team acknowledged this finding and clarified that the team prefers to keep the current implementation, even though it's still safe without the reset.

PR#970: DYNAMIC TICKARRAY

[P5-I-01] Redundant condition

Identified in commit `4007aac`.

The `get_offset` function implements a div_floor feature. If the condition `(r > 0 && rhs < 0) || (r < 0 && rhs > 0)` is true, it indicates that the result is a negative number, and an additional subtraction is applied to round it down.

```
/* programs/whirlpool/src/state/tick_array.rs */
088 | fn get_offset(tick_index: i32, start_tick_index: i32, tick_spacing: u16) -> isize {
089 |     // TODO: replace with i32.div_floor once not experimental
090 |     let lhs = tick_index - start_tick_index;
091 |     let rhs = tick_spacing as i32;
092 |     let d = lhs / rhs;
093 |     let r = lhs % rhs;
094 |     let o = if (r > 0 && rhs < 0) || (r < 0 && rhs > 0) {
095 |         d - 1
096 |     } else {
097 |         d
098 |     };
099 |     o as isize
100 | }
```

However, since the parameter `tick_spacing` is always greater than zero, the rhs variable will also always be greater than zero. Therefore, the condition `(r > 0 && rhs < 0)` is redundant.

It is recommended to remove the redundant condition `(r > 0 && rhs < 0)`.

Resolution

The condition `if (r > 0 && rhs < 0) || (r < 0 && rhs > 0)` has been simplified to `if r < 0` by commit orca-so/whirlpools @ 318dc9f.

PR#970: DYNAMIC TICKARRAY

[P5-I-02] Unused function

Identified in commit `4007aac`.

In `dynamic_tick_array.rs`, the function `is_initialized` is intended to check whether the current dynamic tick has been initialized. However, this function is never used.

```
/* programs/whirlpool/src/state/dynamic_tick_array.rs */
036 | pub fn is_initialized(self) -> bool {
037 |     match self {
038 |         DynamicTick::Initialized(_) => true,
039 |         DynamicTick::Uninitialized => false,
040 |     }
041 | }
```

In the implementation of `DynamicTickArray`, initialization checks are instead performed by directly inspecting the target tick's data. Specifically, check whether the byte at the corresponding offset is set to 1.

```
/* programs/whirlpool/src/state/dynamic_tick_array.rs */
193 | let byte_offset = byte_offsets[curr_offset as usize];
194 | let initialized = self.tick_data()[byte_offset] == 1; // DynamicTick::Initialized
```

It is recommended to remove the redundant function.

Resolution

Function `is_initialized` has been removed by orca-so/whirlpools @ 318dc9f.

PR#970: DYNAMIC TICKARRAY

[P5-Q-01] Potential incorrect error code and comment

Identified in commit `3ddf979`.

In `ensure_position_has_enough_rent_for_ticks()`, if the `additional_rent_required` is greater than the `tick_required_rent`, the function returns the error code `LiquidityTooHigh`.

However, this seems to be a case of insufficient rent, not excessive liquidity. The position's liquidity appears unrelated to the account's rent balance.

Additionally, it's not clear what "there wasn't enough rent for the position to begin with" refers to?

Since a position account's data length is immutable, its rent-exempt threshold should be a constant value. We didn't find an execution path for a position account to become rent-insufficient after initialization.

```
/* programs/whirlpool/src/instructions/reset_position_range.rs */
062 | let rent = Rent::get()?;
063 |
064 | let position_rent_required = rent.minimum_balance(Position::LEN);
065 | let all_required_rent = rent.minimum_balance(Position::LEN + 2 * TICK_INITIALIZE_SIZE);
066 | let tick_required_rent = all_required_rent - position_rent_required;
067 |
068 | let position_lamports = position.to_account_info().lamports();
069 | if position_lamports < all_required_rent {
070 |     // If the position doesn't have enough rent, we need to transfer more SOL from the funder to
↪  the position
071 |     let additional_rent_required = all_required_rent - position_lamports;
072 |
073 |     // Safeguard
074 |     if additional_rent_required > tick_required_rent {
075 |         // This means that there wasn't enough liquidity for the position to begin with
076 |         return Err(ErrorCode::LiquidityTooHigh.into());
077 |     }
```

Resolution

The error code was replaced by an `unreachable` macro by the commit orca-so/whirlpools @ 318dc9f.

PR#970: DYNAMIC TICKARRAY

[P5-Q-02] Will only variable-sized tick arrays be supported in the future?

Identified in commit `4007aac`.

Once the variable tick array feature is enabled, whirlpools will support both fixed and variable tick arrays. Compared to fixed tick arrays, variable tick arrays can effectively reduce the rent costs associated with initializing tick arrays. Users are required to pay the additional tick data rent fees when the variable tick array increases in data length.

However, even users who only open positions using fixed tick arrays must still pay these rent fees. Although these fees are refunded when a position is closed, they still impose an additional burden.

Would it be more economical for whirlpools to support only variable tick arrays?

Resolution

The team clarified that it's the intended behavior.

When using only `FixedTickArrays`, ideally, the program should not send rents to the position if it's not needed. However, since the `open_position` instruction doesn't receive the `TickArray` accounts as inputs, their states cannot be checked at that point. Therefore, the team decided to collect the rent uniformly.

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 Orca Management Company, S.A (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.