Arrakis Modular

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Cccz, Security Researcher

September 12, 2025

Contents

DRAFT

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

DRAFT

2 Security Review Summary

Arrakis is a market maker that powers onchain liquidity for token issuers and LPs

From Sep 1st to Sep 7th the Cantina team conducted a review of arrakis-modular on commit hash 2492b96b. The team identified a total of **11** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	2	0
Low Risk	4	2	2
Gas Optimizations	0	0	0
Informational	5	3	2
Total	**11**	**7**	**4**

DRAFT

3 Findings

3.1 Medium Risk

3.1.1 Oracle returns 0 price when the price difference between the token pairs is too large

Severity: Medium Risk

Context: PancakeSwapV3StandardModule.sol#L1408-L1426

Description: In `_checkMinReturn()`, the protocol fetches prices from the Oracle for slippage checks. The price will be an unnormalized price scaled by the token's decimals:

```
function _checkMinReturn(
    bool zeroForOne_,
    uint256 expectedMinReturn_,
    uint256 amountIn_,
    uint8 decimals0_,
    uint8 decimals1_
) internal view {
    if (zeroForOne_) {
        if (
            FullMath.mulDiv(
                expectedMinReturn_, 10 ** decimals0_, amountIn_
            )
                < FullMath.mulDiv(
                    oracle.getPrice0(), PIPS - maxSlippage, PIPS
                )
        ) revert ExpectedMinReturnTooLow();
    } else {
        if (
            FullMath.mulDiv(
                expectedMinReturn_, 10 ** decimals1_, amountIn_
            )
                < FullMath.mulDiv(
                    oracle.getPrice1(), PIPS - maxSlippage, PIPS
                )
        ) revert ExpectedMinReturnTooLow();
    }
}
```

The problem here is that the scaling is insufficient when the price difference between token pairs is too large.

For example, for the PancakeSwap v3 WBTC/GQ token pair, `sqrtPriceX96 == 177325948760630947712450168439650557727`, `getPrice1()` will be `2 192 * 1e18 / 177325948760630947712450168439650557727 2 == 0 (0.199 round down to 0)`.

That is, for WBTC priced at 100,000 USD, any token priced below 0.001 USD (1 WBTC = 1e8 Token) will have the oracle price rounded down to 0. Also note that for popular token pairs like WBTC/USDC(T), precision loss can also cause oracle price to be higher than actual price. For example, when 1 WBTC == 111000 USDC, `getPrice1()` returns 1e8 / 1110000 == 900 (900.9 rounded down to 900), but the corresponding WBTC price is 1e8 / 900 == 111111 USDC.

Recommendation: It is recommended to increase the price scaling.

Arrakis Finance: Fixed in PR 256.

Cantina Managed: Fix verified.

3.1.2 `totalUnderlyingAtPrice` miscomputes fees using hypothetical tick

Severity: Medium Risk

Context: PancakeSwapV3StandardModule.sol#L927-L932

Description: In `PancakeSwapV3StandardModule`, the at-price view path uses a caller-supplied `sqrtPriceX96_` to derive a hypothetical tick and then uses that hypothetical tick to compute fee growth ”inside” the position. In Uniswap v3, fee attribution depends on the real current pool tick because ”inside” is defined as global fee growth minus growth ”below lower” and ”above upper”, and whether `feeGrowthOutside`

DRAFT

represents "below" or "above" flips when the current tick crosses a boundary. See Uniswap v3 Book on swap fees.

Where the hypothetical tick is injected (for at-price view):

- `src/abstracts/PancakeSwapV3StandardModule.sol`:

```
positionUnderlyingNft = PositionUnderlyingV3Nft({
tokenId: tokenId,
nftPositionManager: nftPositionManager,
pool: _pool,
tick: TickMath.getTickAtSqrtRatio(sqrtPriceX96_),  // hypothetical tick
sqrtPriceX96: sqrtPriceX96_                        // hypothetical price
});
(amt0, amt1, f0, f1) = UnderlyingV3.getUnderlyingBalancesNft(positionUnderlyingNft);
```

This hypothetical tick is then used to compute fees:

- `src/libraries/UnderlyingV3.sol`:

```
(fee0, fee1) = _getFeesEarned(GetFeesPayload({
feeGrowthInside0Last: feeGrowthInside0LastX128,
feeGrowthInside1Last: feeGrowthInside1LastX128,
pool: positionUnderlying_.pool,
liquidity: liquidity,
tick: positionUnderlying_.tick,   // hypothetical tick used here
lowerTick: tickLower,
upperTick: tickUpper
}));
```

The branching that defines "inside" fee growth is explicitly tick-dependent:

- `src/libraries/UnderlyingV3.sol`:

```
function _computeFeesEarned(ComputeFeesPayload memory c) private pure returns (uint256 fee) {
unchecked {
    uint256 feeGrowthBelow = (c.tick >= c.lowerTick)
    ? c.feeGrowthOutsideLower
    : c.feeGrowthGlobal - c.feeGrowthOutsideLower;

    uint256 feeGrowthAbove = (c.tick < c.upperTick)
    ? c.feeGrowthOutsideUpper
    : c.feeGrowthGlobal - c.feeGrowthOutsideUpper;

    uint256 feeGrowthInside = c.feeGrowthGlobal - feeGrowthBelow - feeGrowthAbove;
    fee = FullMath.mulDiv(
    c.liquidity,
    feeGrowthInside - c.feeGrowthInsideLast,
    0x100000000000000000000000000000000
    );
}
}
```

If the hypothetical tick places price in a different region than reality (e.g., pretending price is below the range while the actual price is inside), the code subtracts the wrong "below/above" components and produces an incorrect `feeGrowthInside` for "now". Two incorrect flows can occur:

- Misclassification (no underflow): If branches resolve so that `feeGrowthBelow + feeGrowthAbove` `0`, then `feeGrowthInside` `feeGrowthGlobal`. The view effectively attributes the entire global fee growth as "inside," inflating `f0/f1` massively and, by extension, `totalUnderlyingAtPrice` which adds net-of-manager fees to the at-price token amounts.
- Underflow in unchecked math: If branch choices yield `feeGrowthInside < feeGrowthInsideLast` or `feeGrowthBelow + feeGrowthAbove > feeGrowthGlobal`, the unchecked subtractions wrap, turning a negative delta into a huge `uint256`. That will cause an astronomically large "fees" in views.

Fees must be computed using the real current pool tick (`slot0.tick`). Using a hypothetical tick is incorrect for fees and can materially inflate or deflate the fee component of at-price totals.

Proof of Concept: _(See gist f295edd9).

This proof of concept includes two focused tests that isolate how using a hypothetical tick (derived from a caller-supplied `sqrtPriceX96`) to compute fees causes incorrect "inside" fee growth. Both tests mint a

DRAFT

fresh single-range position, then compare fee/total views under the real pool tick vs. a hypothetical tick.

- Test 1: Misclassification (hypothetical tick below range).
  - Setup: Mint one NFT. Read tickLower/upper and the real slot0.tick. Build a hypothetical tick far below tickLower and pass it to `UnderlyingV3.getUnderlyingBalancesNft` via `PositionUnderlyingV3Nft`.
  - Real tick result: Fees are ≈ 0 (fresh mint), at-price totals match `totalUnderlying()`.
  - Hypo tick result: Fees jump to huge values, and at-price totals balloon.
  - Why: The fee computation branches on the provided tick to decide "below/above." With a hypothetical tick far below, common pool state yields:
  - `below = global − outLower` ≈ 0, `above = outUpper` ≈ 0 → `inside` ≈ `global`.
  - The view ends up attributing nearly the entire global fee growth to the position as "inside", inflating fee outputs. This path doesn't necessarily underflow; it's a pure misclassification that treats "all global" as "inside".
- Test 2: Underflow/wrap (hypothetical tick above range).
  - Setup: Same as Test 1, but choose a hypothetical tick far above tickUpper. Compute "below/above" using the hypothetical tick and print a guard: (below + above) > global.
  - Real tick result: Fees ~ 0 (fresh mint).
  - Hypo tick result: `(below + above) > global` holds, so `inside = global − below − above < 0`. In `UnderlyingV3._computeFeesEarned` this subtraction is inside an unchecked block and wraps to a huge uint256, which then yields astronomically large "fees" when multiplied by liquidity and scaled by Q128.
  - Why: With real price inside the range, typical state is `outLower` ≈ `global`, `outUpper` ≈ 0. Using tick > upper flips the branches to:
  - `below = outLower` ≈ `global`, `above = global − outUpper` ≈ `global` → `below + above` ≈ 2 × `global` > `global`.
  - `inside` subtraction goes negative, wrapping to a massive "inside growth" and therefore massive "fees".

This proof of concept proves that fees must be computed relative to the real current tick. Using a hypothetical tick for the fee component can produce an actual underflow in unchecked arithmetic, wrapping to huge values (Test 2). Also, in both cases, the inflated fee component propagates into `totalUnderlyingAtPrice`, which adds fees (net manager cut) to the at-price token amounts, further ballooning the totals.

Recommendation: Compute token amounts at the hypothetical price, but compute fees with the real pool tick:

- Keep using the caller-supplied `sqrtPriceX96_` for amount decomposition (`LiquidityAmounts.getAmountsForLiquidity`).
- Always source `tick` from `IUniswapV3PoolVariant(pool).slot0().tick` for fee growth "inside", or use a helper that does so (e.g., mirror `UnderlyingV3.totalUnderlyingAtPriceWithFees`, which uses hypothetical price only for amounts while using the real tick for fees).

Arrakis Finance: Fixed in PR 260.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Dust liquidity NFTs may block `MetaVaultOwner` from withdrawing funds

Severity: Low Risk

Context: PancakeSwapV3StandardModule.sol#L1140-L1162

Description: When the MetaVaultOwner withdraws funds, the protocol proportionally reduces the liquidity of all NFTs:

DRAFT

```
uint128 liquidity;
(cakeAmountCollected, liquidity) =
    _unstake(modifyPosition_.tokenId);

liquidity = SafeCast.toUint128(
    FullMath.mulDiv(
        liquidity, modifyPosition_.proportion, BASE
    )
);

INonfungiblePositionManager.DecreaseLiquidityParams memory
    params = INonfungiblePositionManager
        .DecreaseLiquidityParams({
        tokenId: modifyPosition_.tokenId,
        liquidity: liquidity,
        amount0Min: 0,
        amount1Min: 0,
        deadline: type(uint256).max
    });

(burn0, burn1) = INonfungiblePositionManager(
    nftPositionManager
).decreaseLiquidity(params);
```

The problem here is that `INonfungiblePositionManager.decreaseLiquidity()` requires the decreased liquidity to be greater than 0. If the protocol has dusty liquidity NFTs, then when the `MetaVaultOwner` performs the partial withdrawal, the calculated decreased liquidity will be rounded down to 0, causing the withdrawal to fail:

```
function decreaseLiquidity(DecreaseLiquidityParams calldata params)
    external
    payable
    override
    isAuthorizedForToken(params.tokenId)
    checkDeadline(params.deadline)
    returns (uint256 amount0, uint256 amount1)
{
    require(params.liquidity > 0);
```

Recommendation: It is recommended to skip the `INonfungiblePositionManager.decreaseLiquidity()` function call when the decreased liquidity is 0.

Arrakis Finance: Fixed in PR 257.

Cantina Managed: Fix verified.

3.2.2 Missing slippage checks in withdraw function

Severity: Low Risk

Context: PancakeSwapV3StandardModule.sol#L1155-L1156

Description: The withdraw flow reduces each staked NFT's liquidity via `_decreaseLiquidity`, which constructs a Pancake NFPM call with `amount0Min: 0`, `amount1Min: 0`, and `deadline: type(uint256).max`. This removes per-position price protection on burns and leaves the transaction valid indefinitely. Concretely, `withdraw` iterates `tokenIds` and calls `_decreaseLiquidity(modifyPosition)` to burn a proportion of each position and collect amounts for the receiver. Inside `_decreaseLiquidity`, the params are built with zero mins and a max deadline:

```
INonfungiblePositionManager.DecreaseLiquidityParams memory
    params = INonfungiblePositionManager
        .DecreaseLiquidityParams({
        tokenId: modifyPosition_.tokenId,
        liquidity: liquidity,
        amount0Min: 0,
        amount1Min: 0,
        deadline: type(uint256).max
    });
```

A searcher can manipulate spot price within the inclusion block (or exploit the unbounded deadline if the tx lingers) to push the realized burn outputs to be highly one-sided near range edges. Total value at

DRAFT

execution time is broadly conserved, but the token composition delivered to the receiver can be adversarial and force additional swaps or cause portfolio drift. This issue is not present in `rebalance`, which enforces aggregate min-burn/min-deposit checks, however `withdraw` applies no equivalent constraints.

Recommendation: Add slippage bounds and a bounded deadline to withdraw. Derive per-NFT `amount0Min`/`amount1Min` from a robust reference (e.g., pool TWAP via your oracle and a configured tolerance, possibly reusing `maxSlippage`) and use those in `DecreaseLiquidityParams`. Alternatively, accept aggregate `min0Out`/`min1Out` for the whole withdraw and enforce them post-loop.

Arrakis Finance: Acknowledged. This a known issue, but we don't find any attacks of this type that can be profitable for attacker to run. We concluded that a swap happening just before withdraw action that can change the amount of token0/token1 that is withdrawn, can only be a normal swap. And token issuers are ok with that.

Cantina Managed: Acknowledged.

3.2.3 Unbounded iteration over all tokenIds can hit block gas limit

Severity: Low Risk

Context: PancakeSwapV3StandardModule.sol#L735

Description: The `PancakeSwapV3StandardModulePrivate` contract iterates over the entire `_tokenIds` set in several public functions without any bound or batching, for example in `withdrawManagerBalance`:

```
uint256[] memory tokenIdsList = _tokenIds.values();
for (uint256 i; i < tokenIdsList.length;) {
    (uint256 f0, uint256 f1, uint256 cakeCo) = _collectFees(tokenIdsList[i]);
    // ...
    unchecked { i += 1; }
}
```

As the number of NFT positions grows, these O(n) loops can exceed the block gas limit, causing the call to revert and preventing fee withdrawals, claims and withdrawals that rely on processing all positions in one transaction. Since the set is controlled by strategy actions (mints over time), this can lead to a self-inflicted denial-of-service where maintenance operations (e.g., `withdrawManagerBalance`) become unexecutable on-chain.

Recommendation: Consider enforcing a cap on the number of minted/tracked positions.

Arrakis Finance: Acknowledged. Strategies run by Arrakis backend will never have more than 10 positions at a time.

Cantina Managed: Acknowledged.

3.2.4 The slippage in `_checkMinReturn()` is slightly higher than `maxSlippage`

Severity: Low Risk

Context: PancakeSwapV3StandardModule.sol#L1401-L1427

Description: The `checkMinReturn()` function requires that the swap price must be greater than or equal to the oracle price applied with the `maxSlippage`.

```
function _checkMinReturn(
    bool zeroForOne_,
    uint256 expectedMinReturn_,
    uint256 amountIn_,
    uint8 decimals0_,
    uint8 decimals1_
) internal view {
    if (zeroForOne_) {
        if (
            FullMath.mulDiv(
                expectedMinReturn_, 10 ** decimals0_, amountIn_
            )
                < FullMath.mulDiv(
                    oracle.getPrice0(), PIPS - maxSlippage, PIPS
                )
        ) revert ExpectedMinReturnTooLow();
    } else {
```

DRAFT

```
        if (
            FullMath.mulDiv(
                expectedMinReturn_, 10 ** decimals1_, amountIn_
            )
                < FullMath.mulDiv(
                    oracle.getPrice1(), PIPS - maxSlippage, PIPS
                )
        ) revert ExpectedMinReturnTooLow();
    }
}
```

Therefore, for rounding direction:

- When calculating the swap price, round down.
- When applying `maxSlippage` to the Oracle price, round up.

This slightly undervalues the swap price while overvalues the slippage price, ensuring the slippage does not exceed `maxSlippage`.

For example, considering the WBTC/USDC pair, `1 WBTC == 110000 USDC`, `zeroForOne == false`, Oracle price is `1e8 / 110000 == 909`, `maxSlippage == 1%`, Slippage price is `909 * 99% == 899 (899.91 rounded down to 899)`. However, `899` actually corresponds to `maxSlippage == (1 - 899/909) == 1.1%`, resulting in actual slippage exceeding the `maxSlippage`.

Recommendation: It is recommended to round up when calculating the slippage price.

```
  function _checkMinReturn(
      bool zeroForOne_,
      uint256 expectedMinReturn_,
      uint256 amountIn_,
      uint8 decimals0_,
      uint8 decimals1_
  ) internal view {
      if (zeroForOne_) {
          if (
              FullMath.mulDiv(
                  expectedMinReturn_, 10 ** decimals0_, amountIn_
              )
-                 < FullMath.mulDiv(
+                 < FullMath.mulDivRoundingUp(
                      oracle.getPrice0(), PIPS - maxSlippage, PIPS
                  )
          ) revert ExpectedMinReturnTooLow();
      } else {
          if (
              FullMath.mulDiv(
                  expectedMinReturn_, 10 ** decimals1_, amountIn_
              )
-                 < FullMath.mulDiv(
+                 < FullMath.mulDivRoundingUp(
                      oracle.getPrice1(), PIPS - maxSlippage, PIPS
                  )
          ) revert ExpectedMinReturnTooLow();
      }
  }
```

Arrakis Finance: Fixed in PR 261.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 `validateRebalance` is never enforced by `rebalance`

Severity: Informational

Context: PancakeSwapV3StandardModule.sol#L935-L938

Description: The `PancakeSwapV3StandardModule.validateRebalance` is an external view function that compares the pool's price to an oracle and reverts if deviation exceeds a threshold. However, it is not invoked inside the modules' own `rebalance` function. Currently, `ArrakisStandardManager.rebalance` calls `module.validateRebalance` before and after executing the module's `rebalance` payloads, which means

DRAFT

the deviation guard lives only at the manager layer. This creates a policy gap: The module will happily process burns, mints and swaps without any local enforcement if rebalance is ever triggered through a code path that omits the manager's pre/post checks, or if future integrations call the module directly.

Recommendation: Enforce the deviation check within the `PancakeSwapV3StandardModule.rebalance` function before any state-changing actions. Persist a trusted oracle and `maxDeviationPIPS` in module state at initialization and invoke the `validateRebalance` function at the top of `rebalance`.

Arrakis Finance: Acknowledged. The `rebalance` function of the `PancakeSwapV3StandardModule` contract can only be called by the manager through his `rebalance` function. And this `Manager.rebalance` function already calls `PancakeSwapV3StandardModule.validateRebalance` before and after the module `PancakeSwapV3StandardModule.rebalance` function call.

Arrakis Finance: Acknowledged. The `rebalance` function of `PancakeSwapV3StandardModule` can only be called by the manager through his rebalance function. And this rebalance function as a `validateRebalance` function calls before and after module `rebalance` function call.

Cantina Managed: Acknowledged.

3.3.2 Missing require check in `_mint` function

Severity: Informational

Context: PancakeSwapV3StandardModule.sol#L1340

Description: The `_mint` helper calls the NFPM with whatever recipient is provided in `MintParams` and only afterwards tries to stake the newly minted NFT into `MasterChefV3`:

- It approves `amount0Desired/amount1Desired` and calls `INonfungiblePositionManagerPancake(nftPositionManager).mint(params_)`.
- At the end, it unconditionally executes: `IERC721(nftPositionManager).safeTransferFrom(address(this), masterChefV3, tokenId, "")`.

If `params_.recipient != address(this)`, the NFT is minted to a different address, so the module is not the owner and the `safeTransferFrom(address(this), ...)` call reverts. Funds are safe (full revert), but gas is wasted and rebalance becomes brittle to mis-specified params.

Recommendation: Consider adding the following explicit check at the start of `_mint` function:

```
require(params_.recipient == address(this), "RecipientNotModule");
```

Arrakis Finance: Fixed in PR 258.

Cantina Managed: Fix verified.

3.3.3 Missing `LogFund` event emission in `fund()` function

Severity: Informational

Context: PancakeSwapV3StandardModulePrivate.sol#L43-L47

Description: In the `PancakeSwapV3StandardModulePrivate` contract, the `fund` function transfers the specified token amounts from the `depositor` to the module but does not emit the `LogFund` event declared in `IArrakisLPModulePrivate` interface. This deviates from the behavior of other private modules that emit this event upon funding. Off-chain indexers and operational dashboards typically rely on deposit/funding events for accurate accounting, monitoring and audit trails. Without the event, deposits are harder to track, reconcile and alert on.

Recommendation: Emit the `LogFund` event after the transfers and before returning to align with the interface and other modules, for example: `emit LogFund(depositor_, amount0_, amount1_)`.

Arrakis Finance: Fixed in PR 259.

Cantina Managed: Fix verified.

DRAFT

3.3.4 Fund function is payable but explicitly rejects ETH

Severity: Informational

Context: PancakeSwapV3StandardModulePrivate.sol#L47

Description: The `PancakeSwapV3StandardModulePrivate.fund` function is declared `payable` while immediately rejecting any native value via `if (msg.value > 0) revert NativeCoinNotAllowed()`. The function only transfers ERC20 tokens and never handles native currency, so marking it `payable` is contradictory and unnecessary. A non-payable function would automatically revert on `value > 0` without needing an explicit runtime check, slightly simplifying bytecode.

Recommendation: If you can change the interface, update `IArrakisLPModulePrivate.fund` to be non-payable and remove the redundant guard in implementations:

- Change signature to: `function fund(...) external onlyMetaVault whenNotPaused nonReentrant`.
- Remove the `if (msg.value > 0) revert NativeCoinNotAllowed();` line.

Arrakis Finance: Acknowledged. `PancakeSwapV3StandardModulePrivate` is implementing the `IArrakisLP-ModulePrivate` which is common to all private modules, so removing "payable" will make the compilation fail for `PancakeSwapV3StandardModulePrivate`.

Cantina Managed: Acknowledged.

3.3.5 Check `swapPayload.router` is not `token0` or `token1`

Severity: Informational

Context: PancakeSwapV3StandardModule.sol#L573-L580

Description: To avoid unsafe external calls when swapping within the `rebalance()` function, the protocol limits `swapPayload.router`:

```
if (
    params_.swapPayload.router == address(metaVault)
        || params_.swapPayload.router == nftPositionManager
        || params_.swapPayload.router == masterChefV3
        || params_.swapPayload.router == CAKE
) {
    revert WrongRouter();
}

{
    params_.swapPayload.router.functionCall(
        params_.swapPayload.payload
    );
}
```

Since the contract holds `token0` and `token1`, external calls to `token0/1.approve()` are also unsafe. Therefore, it is recommended to check that `swapPayload.router` cannot be `token0` or `token1`.

Note that it is not exploitable now, since the swap requires that `expectedMinReturn` cannot be 0 (or `maxSlippage` to be 100%), otherwise it will fail the `_checkMinReturn()` check, which means external calls must return at least 1 wei of tokens.

```
function _checkMinReturn(
    bool zeroForOne_,
    uint256 expectedMinReturn_,
    uint256 amountIn_,
    uint8 decimals0_,
    uint8 decimals1_
) internal view {
    if (zeroForOne_) {
        if (
            FullMath.mulDiv(
                expectedMinReturn_, 10 ** decimals0_, amountIn_
            )
                < FullMath.mulDiv(
                    oracle.getPrice0(), PIPS - maxSlippage, PIPS
                )
        ) revert ExpectedMinReturnTooLow();
```

DRAFT

Recommendation: It is recommended to check that `swapPayload.router` cannot be `token0` or `token1`.

Arrakis Finance: Fixed in PR 262.

Cantina Managed: Fix verified.

DRAFT