Sorella Angstrom Security Review

Auditors

Desmond Ho, Lead Security Researcher

Milotruck, Security Researcher

Report prepared by: Lucas Goiriz

October 31, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

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.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Sorella Angstrom according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

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

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of xx days in total, Sorella Labs engaged with Spearbit to review the sorella-angstrom protocol. In this period of time a total of **25** issues were found.

Summary

Project Name	Sorella Labs
Repository	sorella-angstrom
Commit	107382...f485
Type of Project	MEV Protection, AMM
Audit Timeline	Oct 14th to Oct 28th
Fix period	Oct 29th - Oct 31st

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	5	5	0
Medium Risk	3	2	1
Low Risk	6	3	3
Gas Optimizations	1	1	0
Informational	10	5	5
Total	**25**	**16**	**9**

5 Findings

5.1 High Risk

5.1.1 Incorrect fee growth initialisation causes incorrect reward distribution

Severity: High Risk

Context: PoolUpdates.sol#L74-L92

Description: The initialisation of the `rewardGrowthOutside` of a tick differs from Uniswap's. It is set to maintain the following invariant:

```
for all ticks: if (tick > current_tick) then
    for all ticks: if (tick' > tick) then
        growthOutside[tick] >= growthOutside[tick']
```

However, it is possible to re-initialise the `rewardGrowthOutside` when it shouldn't. For instance, one could create multiple positions with the same uninitialised lower tick, but different upper ticks that may have different growth outside values in the same block. This results in being able to create a new position being initialised with non-zero rewards.

Proof of Concept: Insert into `PoolUpdates.t.sol`:

```
function test_IncorrectFeeGrowthInside() public {
    uint128 liq1 = 8.2e21;
    address lp1 = makeAddr("lp_1");
    handler.addLiquidity(lp1, 120, 180, liq1);
    handler.addLiquidity(lp1, 120, 300, liq1);
    handler.addLiquidity(lp1, 120, 360, liq1);

    uint128 amount1 = 23.872987e18;
    handler.rewardTicks(re(TickReward({tick: 180, amount: amount1})));
    bumpBlock();
    handler.rewardTicks(re(TickReward({tick: 300, amount: amount1})));
    bumpBlock();

    uint128 liq2 = 0.64e21;
    address lp2 = makeAddr("lp_2");
    handler.addLiquidity(lp2, 60, 300, liq2);
    handler.addLiquidity(lp2, 60, 180, liq2);
    uint256 positionRewardsUpperTick300 = positionRewards(lp2, 60, 300, liq2);
    uint256 positionRewardsUpperTick180= positionRewards(lp2, 60, 180, liq2);

    assertEq(positionRewardsUpperTick180, 0);
    // the position with 300 as the upper tick was initialised with non-zero rewards
    assertGt(positionRewardsUpperTick300, 0);
}
```

Recommendation: Consider initialising to the way Uniswap does, to the global growth outside value. However, this will affect the way `feeGrowthInside` and `pastRewards` is calculated too.

Sorella: Fixed in commit 26db3bbd, requiring several changes:

- Initializing tick growth-outside-accumulators that are below or at the current to the global accumulator (just as is done in Uniswap V4).
- Making sure relevant arithmetic is `unchecked` to allow for wrapping based on how relevant accumulators are initialized.

  Swap out Solady's "WAD Math" (fixed point arithmetic with a base of `10**18`) with so-called "x128" math, fixed point arithmetic with a base of `2**128`. This was necessary due to large rounding errors diminishing LP rewards discovered during further testing.

Spearbit: Fixed. Consider having `cumulativeGrowth` and `globalGrowth` math to be unchecked as well, to be aligned with Uniswap. Accumulation overflows once reward accumulation exceeds `type(uint128).max`, though it take a long time before it occurs.

5.1.2 Incorrect `extraFeeAsset0` fee accounting

Severity: High Risk

Context: UserOrderBuffer.sol#L189-L205

Description: The application of `extraFeeAsset0` is inconsistent depending on the direction & specification of the swap:

- `zeroForOne`:
  - Exact input: `extraFeeAsset0` is subtracted from `quantityIn`, which means the less `asset0` is taken from the user, although less `asset1` is given.
  - Exact output: For the specified `asset1`, `extraFeeAsset0` is subtracted from the required `asset0` payable. Likely to cause the order to be excluded from the bundle because it would cause the `asset0` bundle delta to be negative and thus the bundle transaction to revert with the `BundleChangeNetNegative()` error.
- `oneForZero`:
  - Exact input: Correctly implemented.
  - Exact output: The user receives less than the specified quantity, but will pay less as well.

Recommendation: The implementation can be made cleaner by doing the exact input / output conversion first, then applying `extraFeeAsset0`:

```
if (variant.specifyingInput()) {
    quantityIn = AmountIn.wrap(quantity);
    quantityOut = price.convert(quantityIn);
} else {
    quantityOut = AmountOut.wrap(quantity);
    quantityIn = price.convert(quantityOut);
}

if (variant.zeroForOne()) {
    quantityIn = quantityIn + AmountIn.wrap(extraFeeAsset0);
} else {
    quantityOut = quantityOut - AmountOut.wrap(extraFeeAsset0);
}
```

Sorella: Fixed as suggested in commit ca4e7373.

Spearbit: Verified, the recommended fix was implemented.

5.1.3 `SwapCallLib.call()` incorrectly uses `return()` instead of `revert()` on failed calls

Severity: High Risk

Context: SwapCall.sol#L61-L67

Description: When a swap call to UniswapV4 fails in `SwapCallLib.call()`, the `return` opcode is used instead of reverting:

```
let success :=
    call(gas(), uni, 0, add(self, CALL_PAYLOAD_START_OFFSET), CALL_PAYLOAD_CD_BYTES, 0, 0)
if iszero(success) {
    let free := mload(0x40)
    returndatacopy(free, 0, returndatasize())
    return(0, returndatasize())
}
```

This terminates execution but all existing state changes persist, which makes it possible for wrongly submitted bundles to not revert during execution.

Recommendation: Revert whenever a swap fails instead:

```
  if iszero(success) {
      let free := mload(0x40)
      returndatacopy(free, 0, returndatasize())
-     return(0, returndatasize())
+     revert(0, returndatasize())
  }
```

Sorella: Fixed in commit 152507f2.

Spearbit: Verified, the recommended fix was implemented.

5.1.4 Keys for `_balances` mapping are inverted in `_settleOrderIn()` and `_settleOrderOut()`

Severity: High Risk

Context: Settlement.sol#L25, Settlement.sol#L124, Settlement.sol#L136

Description: In the `Settlement` contract, `_balances` is declared as a mapping of `asset => owner => balance`:

```
mapping(address asset => mapping(address owner => uint256 balance)) internal _balances;
```

However, `_settleOrderIn()` and `_settleOrderOut()` swap the `asset` and `owner` addresses when accessing the `_balances` maping:

```
_balances[from][asset] -= amount;
```

```
_balances[to][asset] += amount;
```

This makes it impossible to use internal balances when handling orders.

Recommendation: In `_settleOrderIn()` and `_settleOrderOut()`, swap the `asset` and `owner` addresses

```
- _balances[from][asset] -= amount;
+ _balances[asset][from] -= amount;
```

```
- _balances[to][asset] += amount;
+ _balances[asset][to] += amount;
```

Sorella: Fixed in commit bdb7b523.

Spearbit: Verified, the recommended fix was implemented.

5.1.5 `deposit()` can be used to inflate a user's balance for not-yet-deployed tokens

Severity: High Risk

Context: Settlement.sol#L33-L36, Settlement.sol#L40-L43

Description: In the `Settlement` contract, both `deposit()` functions transfer in assets from the user and adds to their internal balance:

```
function deposit(address asset, uint256 amount) external {
    asset.safeTransferFrom(msg.sender, address(this), amount);
    _balances[asset][msg.sender] += amount;
}
```

```
function deposit(address asset, address to, uint256 amount) external {
    asset.safeTransferFrom(msg.sender, address(this), amount);
    _balances[asset][to] += amount;
}
```

Both functions use Solady's `SafeTransferLib` to handle token transfers. However, Solady's `SafeTransferLib` does not check if the token has code before when performing any ERC-20 operations, as documented in the comments:

```
/// - For ERC20s, this implementation won't check that a token has code,
///   responsibility is delegated to the caller.
```

If `safeTransferFrom()` is called on an address with no code, it will not revert.

As such, an attacker can inflate their balance in the protocol if they know a token's address ahead of time. For example, an attacker could front-runn a token deployment to call `deposit()` before the token is actually deployed.

Adding this test to `Settlement.t.sol` demonstrates that calling `deposit()` with `asset` as an address without code is possible:

```
function test_depositWithoutCode() public {
    address user = makeAddr("user");
    address asset = address(0xdeadbeef);

    vm.prank(user);
    angstrom.deposit(asset, type(uint256).max);

    assertEq(rawGetBalance(address(angstrom), asset, user), type(uint256).max);
}
```

Recommendation: In both `deposit()` functions, check that the `asset` address has code.

Sorella: Fixed. Upon bubbling this issue up to the Solady maintainers they implemented a fix in e9c03bf4. Angstrom upgraded its Solady dependency to the commit right after in db7f4ef3, replacing the previously implemented hotfix in (the `_safeTransferFrom` method).

Spearbit: Verified. Solady now checks for code existence in `SafeTransferLib`, making this issue no longer possible.

5.2 Medium Risk

5.2.1 `HookBufferLib.readFrom()` reads dirty bytes from free memory

Severity: Medium Risk

Context: HookBuffer.sol#L45-L61

Description: In `HookBufferLib.readFrom()`, data for the hook is copied into memory at offset `memPtr+0x50`, where `memPtr` is the free memory pointer. Afterwards, the hook address is read from `memPtr + 0x44`, as shown below:

```
let hookAddr := mload(add(memPtr, 0x44))
```

However, since `memPtr+0x44` to `memPtr+0x50` hasn't been written to yet, it could contain non-zero bytes used by previous operations. Should this occur, the upper 12 bytes of `hookAddr` will be dirty and could corrupt the `memPtr` stored in `HookBuffer hook` subsequently:

```
hook :=
    or(
        shl(HOOK_MEM_PTR_OFFSET, memPtr),
        or(shl(HOOK_ADDR_OFFSET, hookAddr), add(payloadLength, 0x64))
    )
```

As seen from above, the upper 12 bytes of `hookAddr` will be OR-ed with `memPtr`, causing the hook pointer to be much larger than it should be.

The following PoC demonstrates how `memPtr` is corrupted when `memPtr + 0x44` contains non-zero bytes beforehand:

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

import "forge-std/Test.sol";
import "src/types/CalldataReader.sol";
import "src/types/HookBuffer.sol";

contract HookBufferTest is Test {
    uint256 internal constant HOOK_ADDR_OFFSET = 32;
    uint256 internal constant HOOK_MEM_PTR_OFFSET = 192;
    uint256 internal constant HOOK_LENGTH_MASK = 0xffffffff;

    function getHookBuffer(bytes calldata data) external {
```

```
        CalldataReader reader = CalldataReaderLib.from(data);

        // Fill free memory pointer+0x44 with garbage
        assembly {
            let memPtr := mload(0x40)
            mstore(add(memPtr, 0x44), 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff)
        }

        // Get hook
        (, HookBuffer hook, ) = HookBufferLib.readFrom(reader, false);

        uint256 memPtr;
        address hookAddr;
        assembly {
            memPtr := shr(HOOK_MEM_PTR_OFFSET, hook)
            hookAddr := shr(HOOK_ADDR_OFFSET, hook)
        }

        // memPtr is corrupted by garbage, since memPtr | 0xffffffffffffffff is stored
        assertEq(memPtr, 0xffffffffffffffff);
        assertEq(hookAddr, 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    }

    function test_readFrom() public {
        address hook = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
        bytes memory hookData = abi.encodePacked(hook, "AAAAAAAA");

        this.getHookBuffer(abi.encodePacked(uint24(hookData.length), hookData));
    }
}
```

`SignatureLib.readAndCheckEcdsa()` uses free memory to store 65 signature bytes. As a result, if `HookBufferLib.readFrom()` is called after `SignatureLib.readAndCheckEcdsa()` without any memory allocation in-between, the hook pointer returned by `HookBufferLib.readFrom()` will be corrupted.

This will cause `HookBuffer.tryTrigger()` to revert with `EvmError: MemoryLimitOOG` when trying to read the corrupted hook pointer, making it impossible to execute any user order with a hook. For example, consider two user orders:

1. First user order calls `SignatureLib.readAndCheckEcdsa()` to verify the signature.
2. Second user order contains a hook.

When attempting to execute the hook in the second order, it will revert.

The following imports and test can be added to `Settlement.t.sol` to demonstrate this:

```
import {OrderMeta, ExactFlashOrder} from "test/_reference/OrderTypes.sol";
import {UserOrder, UserOrderLib} from "test/_reference/UserOrder.
```

```
function test_userOrder() public {
    // Give searcher sufficient assetIn
    address assetIn = assets[3];
    MockERC20(assetIn).mint(searcher.addr, 10e18);

    // Enable pool for (assetIn, otherAsset)
    enablePool(assetIn, otherAsset);

    // Create two exact flash orders to swap 1e18 and 2e18 assets respectively
    ExactFlashOrder memory order1;
    ExactFlashOrder memory order2;
    order1.amount = 1e18;
    order2.amount = 2e18;
    order1.exactIn = order2.exactIn = true;
    order1.assetIn = order2.assetIn = assets[3];
    order1.assetOut = order2.assetOut = otherAsset;
    order1.hook = order2.hook = address(this);
    order1.validForBlock = order2.validForBlock = uint64(block.number);

    // Sign both orders
    sign(searcher, order1.meta, erc712Hash(domainSeparator, order1.hash()));
    sign(searcher, order2.meta, erc712Hash(domainSeparator, order2.hash()));

    // Create bundle
    Bundle memory bundle;
```

```
    bundle.addPair(assetIn, otherAsset);
    bundle.userOrders = new UserOrder[](2);
    bundle.userOrders[0] = UserOrderLib.from(order1);
    bundle.userOrders[1] = UserOrderLib.from(order2);

    // Execute orders
    bytes memory payload = bundle.encode(rawGetConfigStore(address(angstrom)));
    vm.prank(validator);
    angstrom.execute(payload);
}

function compose(address, bytes calldata) external pure returns (uint32) {
    return 0x24a2e44b;
}
```

Recommendation: Mask the `hookAddr` as such:

```
- let hookAddr := mload(add(memPtr, 0x44))
+ let hookAddr := and(mload(add(memPtr, 0x44)), 0xffffffffffffffffffffffffffffffffffffffff)
```

Sorella: Fixed in commit 6fb96ead.

Spearbit: Verified, the upper 12 bytes of `hookAddr` are now cleared.

5.2.2 `Pair.getSwapInfo()` returns wrong prices for asset conversion

Severity: Medium Risk

Context: Pair.sol#L171-L177

Description: `Pair.getSwapInfo()` returns `price0Over1` when `zeroToOne = true`, and `price1Over0` when `zeroToOne = false`:

```
assembly ("memory-safe") {
    let offsetIfZeroToOne := shl(5, zeroToOne)
    assetIn := mload(add(self, xor(offsetIfZeroToOne, 0x20)))
    assetOut := mload(add(self, offsetIfZeroToOne))
    priceOutVsIn := mload(add(self, add(PAIR_PRICE_10_OFFSET, offsetIfZeroToOne)))
    oneMinusFee := sub(ONE_E6, mload(add(self, PAIR_FEE_OFFSET)))
}
```

However, the returned price does not match the asset calculations in `UserOrderBuffer.loadAndComputeQuantity()`:

```
if (variant.zeroForOne()) {
    if (variant.specifyingInput()) {
        quantityIn = AmountIn.wrap(quantity - extraFeeAsset0);
        quantityOut = price.convert(quantityIn);
    } else {
        quantityOut = AmountOut.wrap(quantity);
        quantityIn = price.convert(quantityOut) - AmountIn.wrap(extraFeeAsset0);
    }
} else {
    if (variant.specifyingInput()) {
        quantityIn = AmountIn.wrap(quantity);
        quantityOut = price.convert(quantityIn) - AmountOut.wrap(extraFeeAsset0);
    } else {
        quantityOut = AmountOut.wrap(quantity - extraFeeAsset0);
        quantityIn = price.convert(quantityOut);
    }
}
```

For example:

- If `asset1` is WBTC and `asset0` is USDT:
  - `price1Over0` would be `(1.0e8) * 1e27 / (65_000e6) = ~1.538e24`.
  - `price0Over1` would be `1e29`.
- Assume that:
  - `zeroForOne` and `specifyingInput` are both true, meaning the user swaps USDT to WBTC.

  - The amount of USDT to swap is specified as `quantity = 65_000e6`.
- `PairLib.getSwapInfo()` would return `price0Over1`.
- The calculation above be `quantityOut = quantityIn * price0Over1 / 1e27 = 65_000e6 * 1e29 / 1e27 = 65_000e8`.

This returns 65,0000 WBTC for 65,000 USDT, which is incorrect. The correct calculation would be to use `price1Over0` instead.

Recommendation: In `Pair.readFromAndValidate()`, renaming the offsets to `PAIR_PRICE_1_TO_0_OFFSET` and `PAIR_PRICE_0_TO_1_OFFSET` for better clarity and swap the loaded prices:

```
- mstore(add(raw_memoryOffset, PAIR_PRICE_10_OFFSET), price1Over0)
- mstore(add(raw_memoryOffset, PAIR_PRICE_01_OFFSET), price0Over1)
+ mstore(add(raw_memoryOffset, PAIR_PRICE_1_TO_0_OFFSET), price0Over1)
+ mstore(add(raw_memoryOffset, PAIR_PRICE_0_TO_1_OFFSET), price1Over0)

- priceOutVsIn := mload(add(self, add(PAIR_PRICE_10_OFFSET, offsetIfZeroToOne)))
+ priceOutVsIn := mload(add(self, add(PAIR_PRICE_1_TO_0_OFFSET, offsetIfZeroToOne)))
```

Alternatively, swap the returned price in `Pair.getSwapInto()`:

```
  assembly ("memory-safe") {
      let offsetIfZeroToOne := shl(5, zeroToOne)
      assetIn := mload(add(self, xor(offsetIfZeroToOne, 0x20)))
      assetOut := mload(add(self, offsetIfZeroToOne))
-     priceOutVsIn := mload(add(self, add(PAIR_PRICE_10_OFFSET, offsetIfZeroToOne)))
+     priceOutVsIn := mload(add(self, sub(PAIR_PRICE_01_OFFSET, offsetIfZeroToOne)))
      oneMinusFee := sub(ONE_E6, mload(add(self, PAIR_FEE_OFFSET)))
  }
```

Sorella: Fixed in commit dc40ec09.

Spearbit: Verified, the correct price is now returned in `Pair.getSwapInfo()`. Additionally, the commit contains a fix for `ConfigEntryLib.feeInE6()` returning dirty upper bytes.

5.2.3 Bundles with `PermitSubmitterHook` can be forced to revert with permit front-running

Severity: Medium Risk

Context: Angstrom.sol#L250

Description: When users submit user orders, they are allowed to specify a custom hook to be called when their order is executed. One of these hooks is `PermitSubmitterHook`, which uses ERC-20 permit to grant approvals before transferring tokens from the user.

However, if a node calls `execute()` with multiple user orders and one of them calls `PermitSubmitterHook`, anyone can force the bundle to revert by front-running it and directly calling `permit()` with the signature beforehand. This will cause `PermitSubmitterHook.compose()` to revert when attempting to call `permit()` with the same signature, since it has already been used.

This pattern is described in more detail innTrust-security's Permission denied post.

Recommendation: Wrap all `permit()` calls in `PermitSubmitterHook.compose()` in a `try-catch`.

Sorella: Acknowledged.

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 Incorrect Tick Compression Adjustment in `isInitialized()`

Severity: Low Risk

Context: IUniV4.sol#L180

Description: `(int16 wordPos, uint8 bitPos) = TickLib.position(TickLib.compress(tick, tickSpacing) - 1);` is used to get the word and bit positions after compressing the tick by `tickSpacing`. However, there is an incorrect offset by 1, resulting in an incorrect position calculation.

Recommendation:

```
- (int16 wordPos, uint8 bitPos) = TickLib.position(TickLib.compress(tick, tickSpacing) - 1);
+ (int16 wordPos, uint8 bitPos) = TickLib.position(TickLib.compress(tick, tickSpacing));
```

Sorella: Fixed in commit 8b203818.

Spearbit: Verified, the recommended fix was implemented.

5.3.2 Potential DoS Risk with Excessive Initialized Ticks

Severity: Low Risk

Context: GrowthOutsideUpdater.sol#L105

Description: There is a potential Denial of Service (DoS) risk if there are too many initialised ticks to iterate through, which could cause transactions to exceed the block gas limit. This would be more prevalent for pools with low tick spacings, and for ticks that are located further away from the current tick.

Recommendation: Rewarders should be aware of the above limitation when determining what liquidity ranges to reward.

Sorella: Acknowledged. With proper documentation this should be a non issue considering the system at large because:

1. With rewards we aim to reward ticks whose liquidity was used for swapping, therefore these costs should be proportional and roughly limit each other.
2. The gas cost for any attempted operation is computed and split pro-rata among the users who are settled in a given bundle, therefore the willingness of users to pay for gas will be a natural limiting factor.

Documented under known issues in commit 14b8df17.

Spearbit: Acknowledged.

5.3.3 Potential zero reward allocation if reward tick is incorrectly specified

Severity: Low Risk

Context: GrowthOutsideUpdater.sol#L97-L99

Description: Reward distributors should be aware of a footgun, that it's possible to have non-zero `rewardTotal` but zero reward distributed if the `rewardTick` happens to be the upper tick of the highest position (because the upper tick is exclusive of the position).

Note another instance of this occurs when `currentOnly` is `true` and there is 0 active liquidity (ie. `getPoolLiquidity()` = 0).

Proof of Concept: Insert into `PoolUpdates.t.sol`.

```
function test_zeroRewardDistributed() public {
  uint128 liq1 = 8.2e21;
  address lp1 = makeAddr("lp_1");
  handler.addLiquidity(lp1, -120, 120, liq1);

  uint128 amount1 = 23.872987e18;
  // reward upper tick
  handler.rewardTicks(re(TickReward({tick: 120, amount: amount1})));
  uint256 lpRewards = positionRewards(lp1, -120, 120, liq1);
  assertEq(lpRewards, 0);
}

function test_zeroRewardDistributed_currentOnly() public {
  uint128 liq1 = 8.2e21;
  address lp1 = makeAddr("lp_1");
  handler.addLiquidity(lp1, -180, -120, liq1);

  uint128 amount1 = 23.872987e18;
  // reward upper tick
  handler.rewardTicks(re(TickReward({tick: -120, amount: amount1})));
  uint256 lpRewards = positionRewards(lp1, -180, -120, liq1);
  assertEq(lpRewards, 0);
}
```

Recommendation: Consider checking that the starting `liquidity` and `getPoolLiquidity()` is non-zero.

Sorella: Acknowledged. Since the payload for the contract will be built and validated off-chain by code that we're developing we prefer to have this checks off-chain to save gas. This footgun has been documented in a dedicated "Known Issues" document added in commit 14b8df17.

Spearbit: Acknowledged.

5.3.4 Fee overcharging arising from precision issues

Severity: Low Risk

Context: Pair.sol#L178

Description: The calculation of fees can result in overcharging for swaps involving high-value tokens with low decimal precision (such as WBTC and GUSD).

Proof of Concept:

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

import { PriceAB as PriceOutVsIn, AmountA as AmountOut, AmountB as AmountIn } from "src/types/Price.sol";
import { RayMathLib } from "src/libraries/RayMathLib.sol";
import { Test, console2 as console } from "lib/forge-std/src/Test.sol";

contract FeesPOC is Test {
  function test_overchargingFees() public pure {
    // asset0 = WBTC, asset1 = GUSD
    uint256 qtyAsset0 = 1e5; // 0.001 WBTC
    uint256 price1Over0 = uint(65_000e2) * 1e27 / 1e8; // 0To1
    uint256 feeInE6 = 1;
    price1Over0 = applyFeeInE6(price1Over0, feeInE6);

    PriceOutVsIn price1Over0Wrapped = PriceOutVsIn.wrap(price1Over0);

    AmountIn inQtyAsset0 = AmountIn.wrap(qtyAsset0);
    AmountOut outQtyAsset1 = price1Over0Wrapped.convert(inQtyAsset0);
    uint256 actualQuantityOut = AmountOut.unwrap(outQtyAsset1);
    assertEq(actualQuantityOut, 6499);
  }
}
```

In the proof of concept, 0.001 WBTC = 1e5 (= $65 USD) is swapped for GUSD at 65000 GUSD / BTC. The user receives $64.99 (1 cent taken as fee), but the fee should be $0.000065. Also, because fees are taken in `asset0` (WBTC), there is overcharging, since 1 wei of WBTC is $0.00065.

Recommendation: Consider how fees should be applied for such tokens of low decimals and high values, and in general, tokens with non-standard decimals.

Sorella: Acknowledged. Rounding up looks to be limited to 1 base unit of the asset, so by definition the smallest possible amount. We deem this not worth changing.

Spearbit: Acknowledged.

5.3.5 Multiple orders with the same `orderHash` cannot be executed in the same block

Severity: Low Risk

Context: Angstrom.sol#L151, Angstrom.sol#L240

Description: When an `orderHash` is invalidated, the same `orderHash` cannot be used again in the same block:

```
function _invalidateOrderHash(bytes32 orderHash) internal {
    tuint256 storage executed = alreadyExecuted[orderHash];
    if (executed.get() != 0) revert OrderAlreadyExecuted();
    executed.set(1);
}
```

This is used to prevent signature replay for top-of-block orders and flash user orders.

However, a possible issue with this implementation is that it could block multiple legitimate orders with the same `orderHash`. For example, two user orders from different users could happen to have the `UserOrderBuffer`, as shown below:

- `typeHash = EXACT_FLASH_ORDER_TYPEHASH`.
- `refId = 0`.
- `exactIn_or_minQuantityIn = 1`.
- `quantity_or_maxQuantityIn` as 65,000 USDT.
- `maxExtraFeeAsset0` as 65 USDT.
- `minPrice` as 64,500 USDT / WBTC.
- `useInternal = false`.
- `assetIn` as USDT, `assetOut` as WBTC.
- `recipient = address(0)`.
- `hookDataHash = EMPTY_BYTES_HASH`.
- `nonce_or_validForBlock = block.number`.

Both users submit a flash order to buy 65,000 USDT worth of WBTC and receive it with their address. As a result, both orders have the same `orderHash` and one of the orders cannot be executed.

The same issue is present with ToB orders, since `_invalidateOrderHash()` is used to invalidate hashes of `ToBOrderBuffer` as well.

Recommendation: Consider invalidating order hashes together with the `from` address. For example:

```
- mapping(bytes32 => tuint256) internal alreadyExecuted;
+ mapping(from => bytes32 => tuint256) internal alreadyExecuted;
```

```
- function _invalidateOrderHash(bytes32 orderHash) internal {
-     tuint256 storage executed = alreadyExecuted[orderHash];
+ function _invalidateOrderHash(address from, bytes32 orderHash) internal {
+     tuint256 storage executed = alreadyExecuted[from][orderHash];
      if (executed.get() != 0) revert OrderAlreadyExecuted();
      executed.set(1);
  }
```

Sorella: Fixed in commit be496c77.

Spearbit: Verified, the recommended fix was implemented.

5.3.6 Incorrect EIP-712 typehash for `ToBOrderBuffer`

Severity: Low Risk

Context: ToBOrderBuffer.sol#L15, ToBOrderBuffer.sol#L34

Description: In the `ToBOrderBuffer` struct, `validForBlock` is declared as a `uint64`:

```
uint64 validForBlock;
```

However, in `TOP_OF_BLOCK_ORDER_TYPEHASH`, which is the EIP-712 typehash for `TobOrderBuffer`, encodes `validForBlock` as a `uint256`:

```
"uint256 valid_for_block"
```

Recommendation: Declare `valid_for_block` as `uint64` in the typehash instead:

```
- "uint256 valid_for_block"
+ "uint64 valid_for_block"
```

Sorella: Fixed in commit dfd84843.

Spearbit: Verified, the recommended fix was implemented.

5.4 Gas Optimization

5.4.1 `liquidityGross` extraction can be optimised

Severity: Gas Optimization

Context: IUniV4.sol#L124

Description: Instead of shifting the value left and right, a mask can be applied to extract the lower 128 bits.

Recommendation:

```
- liquidityGross := shr(128, shl(128, packed))
+ liquidityGross := and(packed, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
```

Sorella: Fixed in commit 4e738812.

Spearbit: Verified, the recommended fix was implemented.

5.5 Informational

5.5.1 Redundancies

Severity: Informational

Context: PoolUpdates.sol#L10, PoolUpdates.sol#L21, PoolUpdates.sol#L25, TopLevelAuth.sol#L17-L18, DeltaTracker.sol#L6, SwapCall.sol#L6, ToBOrderBuffer.sol#L4, UserOrderBuffer.sol#L6

Description: Referenced lines are redundant imports or unused events.

Recommendation: Remove the redundancies.

Sorella: Fixed in commit 59fe5ab7.

Spearbit: Verified.

5.5.2 Variable and comment improvements

Severity: Informational

Context: IHooks.sol#L72, IHooks.sol#L88, PoolConfigStore.sol#L156, PoolConfigStore.sol#L158, PoolUpdates.sol#L31, HookBuffer.sol#L42, HookBuffer.sol#L72, PoolRewards.sol#L9, PoolUpdateVariantMap.sol#L11, PoolUpdateVariantMap.sol#L18, TypedDataHasher.sol#L19, UserOrderBuffer.sol#L173-L174

Description: Referenced lines are incorrect comments, variable names or spelling errors.

Recommendation:

```
- liquidty
+ liquidity

- // Increase `totalEntryBytes` by 0x20 if we broke in the loop.
+ // Increase `totalEntryBytes` by 0x20 if we didn't break in the loop.

- incase
+ in case

- underyling
+ underlying

- aloted
+ allotted

- accomodate
+ accommodate

- CURRENTY_ONLY_FLAG
+ CURRENT_ONLY_FLAG

- // Pre-store ERC721 header bytes and domain separator in memory.
+ // Pre-store ERC712 header bytes and domain separator in memory.

- // Partial order.

- hookDatalength
+ hookDataLength
```

Sorella: Fixed in commit 511d9bbe.

Spearbit: Verified.

5.5.3 Code simplification via conditional function references

Severity: Informational

Context: GrowthOutsideUpdater.sol#L60-L62

Description: In the existing code, a conditional check is performed to decide between two similar functions, `_rewardBelow` and `_rewardAbove`. While this works as expected, the current approach involves duplicating the code structure within the conditional block.

Recommendation: The code can be simplified by using a function reference that points to either `_rewardBelow` or `_rewardAbove` based on the condition.

```
function (
   uint256[REWARD_GROWTH_SIZE] storage,
   int24,
   CalldataReader,
   uint128,
   RewardParams memory
) internal returns (CalldataReader, uint256, uint256, uint128) _reward =
   startTick <= pool.currentTick ? _rewardBelow : _rewardAbove;

   (newReader, total, cumulativeGrowth, endLiquidity) =
      _reward(poolRewards.rewardGrowthOutside, startTick, newReader, liquidity, pool);
```

Sorella: Acknowledged. Would lead to more lines of code and is less aesthetic in the opinion of the author.

Spearbit: Acknowledged.

5.5.4 Hook revert reasons don't bubble up

Severity: Informational

Context: PermitSubmitterHook.sol#L69-L72, HookBuffer.sol#L99-L109

Description: Hook calls revert with `InvalidHookReturn()`, but don't bubble up the revert reason, making debugging a little harder.

Proof of concept:

1. Save and apply `git patch`:

```
diff --git a/contracts/src/modules/PermitSubmitterHook.sol
↪  b/contracts/src/modules/PermitSubmitterHook.sol
index ca0ab1b8..0398b9ed 100644
--- a/contracts/src/modules/PermitSubmitterHook.sol
+++ b/contracts/src/modules/PermitSubmitterHook.sol
@@ -9,7 +9,7 @@ import {IDaiPermit} from "../interfaces/IDaiPermit.sol";
import {CalldataReader, CalldataReaderLib} from "../types/CalldataReader.sol";

/// @author philogy <https://github.com/philogy>
-abstract contract PermitSubmitterHook is IAngstromComposable {
+contract PermitSubmitterHook is IAngstromComposable {
    uint256 internal constant ERC2612_INFINITE = 0x00;
    uint256 internal constant ERC2612_SPECIFIC = 0x01;
    uint256 internal constant DAI_INFINITE = 0x02;
diff --git a/contracts/test/types/HookBuffer.t.sol b/contracts/test/types/HookBuffer.t.sol
index 50cb4b51..240eb037 100644
--- a/contracts/test/types/HookBuffer.t.sol
+++ b/contracts/test/types/HookBuffer.t.sol
@@ -10,6 +10,7 @@ import {
} from "../../src/interfaces/IAngstromComposable.sol";
import {Recorder} from "../_mocks/composable/Recorder.sol";
import {SmolReturn} from "../_mocks/composable/SmolReturn.sol";
+import { PermitSubmitterHook } from "src/modules/PermitSubmitterHook.sol";
import {PRNG} from "super-sol/collections/PRNG.sol";

import {console} from "forge-std/console.sol";
@@ -18,10 +19,38 @@ import {console} from "forge-std/console.sol";
contract HookBufferTest is BaseTest {
    Recorder recorder;
    SmolReturn smol;
+    PermitSubmitterHook permitSubmitter;

    function setUp() public {
        recorder = new Recorder();
        smol = new SmolReturn();
+        permitSubmitter = new PermitSubmitterHook();
+    }
+
+    function test_fuzzing_invalidPermitType(uint8 permitType, address from) public {
+        vm.assume(permitType > 2);
+        bytes memory hookPayload = abi.encodePacked(permitType);
+        vm.expectRevert(HookBufferLib.InvalidHookReturn.selector);
+        this._test_fuzzing_invalidPermitType(
+            abi.encodePacked(
+                uint24(hookPayload.length + 20), address(permitSubmitter), hookPayload
+            ),
+            from,
+            hookPayload
+        );
+    }
+
+    function _test_fuzzing_invalidPermitType(
+        bytes calldata data,
+        address from,
+        bytes calldata hookPayload
+    ) external {
+        CalldataReader reader = CalldataReaderLib.from(data);
+        (CalldataReader outReader, HookBuffer hookBuffer, bytes32 hash) =
+            HookBufferLib.readFrom(reader, false);
+        assertEq(hash, keccak256(abi.encodePacked(address(permitSubmitter), hookPayload)), "wrong
↪  hash");
+        assertEq(reader.offset() + 23 + hookPayload.length, outReader.offset());
+        hookBuffer.tryTrigger(from);
    }
```

```
    function test_emptyBytesHash() public pure {
```

2. Run `forge test --mt test_fuzzing_invalidPermitType`:

Recommendation: Consider adopting ERC-7751 used by Uniswap to wrap and bubble up reverts.

Sorella: Acknowledged. Not implemented for the sake of simplicity and gas. Top-level execute can only be triggered by trusted nodes so need for this level of introspection is not warranted. In case an on-chain transaction needs to be debugged such as tenderly or foundry's cast run will show the original error in their traces.

Spearbit: Acknowledged.

5.5.5 Lenient delta accounting may lead to overlooked fees

Severity: Informational

Context: Settlement.sol#L95-L97

Description: The `bundleDeltas` check allows for positive deltas, but the excess may not be clearly accounted for (eg. from `extraAsset0` and gas fees charged), which may not be included in `saving` that's logged. This may complicate accounting and fee distribution.

Recommendation: Consider adding the excess positive delta into the `saving` variable.

Sorella: Fixed in commit 884cfbcd.

Spearbit: Fixed. `bundleDeltas` must now strictly be zero after settlement.

5.5.6 Missing `memory-safe` annotation on assembly block

Severity: Informational

Context: PoolConfigStore.sol#L183, RayMathLib.sol#L20, TickLib.sol#L41, TickLib.sol#L53

Description: The referenced lines are assembly blocks that are memory safe, but do not have the `memory-safe` annotation.

Recommendation: Add the `memory-safe` annotation to the referenced lines.

Sorella: Suggestion implemented for the sake of consistency, including unmentioned cases in commit c8ac941a.

Spearbit: Verified, the recommendation was implemented.

5.5.7 Minor code improvements

Severity: Informational

Context: Angstrom.sol#L162-L167, TopLevelAuth.sol#L79, HookBuffer.sol#L38, Positions.sol#L27

Description/Recommendation:

1. Positions.sol#L27: This line can be removed as the free memory pointer isn't used.
2. HookBuffer.sol#L38: This line can be removed as `hook` will be `0` by default.
3. Angstrom.sol#L162-L167: `buffer.useInternal` can be used instead of calling `variantMap.useInternal()` again:

```
  _settleOrderIn(
-     from, buffer.assetIn, AmountIn.wrap(buffer.quantityIn), variantMap.useInternal()
+     from, buffer.assetIn, AmountIn.wrap(buffer.quantityIn), buffer.useInternal
  );
  _settleOrderOut(
-     to, buffer.assetOut, AmountOut.wrap(buffer.quantityOut), variantMap.useInternal()
+     to, buffer.assetOut, AmountOut.wrap(buffer.quantityOut), buffer.useInternal
  );
```

4. TopLevelAuth.sol#L79: Using `SafeCastLib` when casting `block.number` to `uint64` is unnecessary.

Sorella: Fixed in commits 7bd44470, ade54cf8 and c591be6a.

Spearbit: Verified, the recommended fixes were implemented except the last.

5.5.8 `PoolConfigStore.get()` does not check if `index` is less than the number of configured pools

Severity: Informational

Context: PoolConfigStore.sol#L184-L185

Description: `PoolConfigStore.get()` does not have a `index < self.totalEntries()` check that ensures `index` is less than the total number of entries.

However, there is no impact in the current protocol's implementation:

1. If `index` is greater than the number of entries, the key check afterwards will revert.
2. If `index` is sufficiently large such that `index * 32 + 1` overflows, it will fortunately wrap around to a value that still fulfills `n * 32 + 1`. Therefore, the entry that is read will always be valid.

Note that (2) doesn't hold true if `ENTRY_SIZE` isn't 32; if `ENTRY_SIZE` is not 32, index being so large such that calculating the code offset overflows would be a problem.

Recommendation: Consider checking that `index` is less than the total number of entries in `PoolConfigStore.get()`.

Sorella: Acknowledged. Since this does not currently have any impact and would impose a gas cost on legitimate calls the suggested fix has not been implemented. Instead, a simple test has been added in commit 1e0aabba to watch for any changes in `ENTRY_SIZE` and serve as an explicit reminder of this potential issue.

Spearbit: Acknowledged.

5.5.9 Multiple pools with the same asset pair but different tick spacing cannot be configured in the protocol

Severity: Informational

Context: PoolConfigStore.sol#L147-L155

Description: In `PoolConfigStore`, entries are stored as:

```
bytes27 key | uint16 tickSpacing | uint24 feeInE6
```

where `key` is the last 27 bytes of the hash of `asset0` and `asset1`:

```
// Construct new entry by splicing in the values.
let newEntry :=
    or(
        key,
        or(
            shl(TICK_SPACING_OFFSET, and(tickSpacing, TICK_SPACING_MASK)),
            shl(FEE_OFFSET, and(feeInE6, FEE_MASK))
        )
    )
```

When adding a new entry, if a previous entry with the same key exists, that previous entry will be replaced. This is done by comparing the key of the previous entry to the current key to be added:

```
// Search pool to see if it was already configured, if so replace the entry.
let entriesEnd := add(entryOffset, totalEntryBytes)
for {} lt(entryOffset, entriesEnd) { entryOffset := add(entryOffset, 0x20) } {
    let entry := mload(entryOffset)
    if eq(key, and(entry, KEY_MASK)) {
        mstore(entryOffset, newEntry)
        break
    }
}
```

As such, it is not possible to have multiple entries with the same pair of `asset0` and `asset1`. If `PoolConfigStore.setIntoNew()` is called with the same `asset0` and `asset1`, but a different `tickSpacing` or `feeInE6`, the old entry will be replaced.

Therefore, unlike Uniswap, it is not possible to have multiple pools with the same asset pair but a different tick spacing.

Recommendation: Document this protocol limitation.

Sorella: Acknowledged. This is intentional as we want to ensure liquidity is concentrated in one pool for any given pair. We intend to choose & update tick spacing carefully such that it maximizes positive LP outcomes.

Spearbit: Acknowledged.

5.5.10 Liquidity can still be deposited into pools no longer in `PoolConfigStore`

Severity: Informational

Context: PoolUpdates.sol#L52

Description: In `PoolUpdates`, the protocol's `beforeAddLiquidity()` hook does not check if the Uniswap V4 pool being deposited into exists in `PoolConfigStore`.

As such, even after a pool is removed with `TopLevelAuth.removePool()` or replaced with `TopLevelAuth.configurePool()`, it is still possible to deposit liquidity into the old pool. However, these pools cannot be used to perform swaps in the protocol.

Recommendation: In `beforeAddLiquidity()`, consider checking if the Uniswap V4 pool is stored in `PoolConfigStore`. This can be achieved by making the user pass in the entry's `storeIndex` in the hook's `hookData`, and checking that `currency0`, `currency1` and `tickSpacing` match the entry in `PoolConfigStore`.

Sorella: Acknowledged. We will not be implementing the suggested fix as it would impose a gas cost on LPs adding liquidity for legitimately active pools. While it is a minor footgun the overall savings to liquidity providers should more than balance out any self imposed losses to transaction costs by mistaken users.

Spearbit: Acknowledged.