Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Poolshark Cover

Veridise Inc.

September 1, 2023

► **Prepared For:**

Poolshark Labs
https://www.poolshark.fi/

► **Prepared By:**

Xiangan He
Ben Mariano
Andreea Buterchi

► **Contact Us:** contact@veridise.com

► **Version History:**

Apr. 06, 2023	Initial Draft
Apr. 12, 2023	V1
Apr. 19, 2023	V2
Apr. 27, 2023	V3

Contents

1 Executive Summary

From Mar. 13, 2023 to Apr. 10, 2023, Poolshark Labs engaged Veridise to review the security of the Poolshark Protocol, an Automated Market Maker (AMM) which supports directional liquidity. The review covered the Cover Pool component which enables liquidity providers (LPs) to "cover" or "hedge" their positions. Veridise conducted the assessment over 12 person-weeks, with 3 engineers reviewing code over 4 weeks on commit `0xf8d337b`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Poolshark Labs developers provided the source code of the Poolshark Protocol contracts for review. To facilitate the Veridise auditors' understanding of the code, the Poolshark Labs developers shared a whitepaper and documentation about the Cover Pool and its mechanisms. In general, the documentation was somewhat scant. In particular, the documentation currently does not have a clear description of the user-facing APIs and their intended behavior (i.e., what parameters are expected, what do they represent, etc.). Furthermore, documentation/comments within their code is limited, which is challenging as the core logic is quite complicated and there are many variables with similar names. Developers have done a good job testing their codebase, including tests that achieve almost 100% code coverage. During the audit, the Poolshark Labs developers made several functional changes to the code. This is because the Poolshark Labs developers were simultaneously performing a code refactor and internally reviewing the code while collaborating with external auditors. Due to this, Veridise auditors had to re-acquaint themselves with the modified code over-time and review subsequent bug-fix commits that were different from the original code. All auditing was performed on commit `0xf8d337b` with the exception of any bug fixes that were verified.

Summary of issues detected. The audit uncovered 21 issues, 3 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, several logic errors were found for functionality used to calculate swap amounts (V-ALL-PSH-001 - V-ALL-PSH-003). The Veridise auditors also identified several medium-severity issues, including losses induced by TWAP Oracle attacks (V-ALL-PSH-005), potential overflows on Tick calculations (V-ALL-PSH-004), and liquidity not being handled correctly in partial mints (V-ALL-PSH-007) as well as a number of minor issues. The Poolshark Labs developers fixed most of the issues reported in the audit (including all major ones) and acknowledged the remaining minor issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Poolshark Protocol. Our first suggestion is to increase the modularity of the code. There are multiple functions which comprise hundreds of lines of code – for clarity and future extension, we suggest splitting these into smaller functions with clearly defined tasks. Our second suggestion is to split some of the logic associated with `token0` and `token1`; one common source of confusion in the code for auditors was understanding functions that needed to handle both, usually resulting in ITE statements over a boolean, where the true and false branches

were almost identical modulo a few small changes. We suspect that some of the logic can be separated, which could actually allow more code-reuse by abstracting away the shared behaviors. Finally, we suggest improving naming of variables and functions. As an example, currently the protocol has a function called `validate` that both validates the user inputs and (in our opinion non-intuitively) updates them if they are incorrect. To improve the readability and maintainability of the code, we suggest function and variable names carefully reflect expected behavior. For a full list of recommendations made by the Veridise auditors, check out the detailed issue report.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Poolshark Protocol	`0xf8d337b`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 13 - Apr. 10, 2023	Manual & Tools	3	12 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	3	3
Medium-Severity Issues	7	7
Low-Severity Issues	4	4
Warning-Severity Issues	1	1
Informational-Severity Issues	6	5
TOTAL	21	20

Table 2.4: Category Breakdown.

Name	Number
Logic Error	8
Locked Funds	0
Denial of Service	1
Data Validation	6
Maintainability	5
Missing/Incorrect Events	0
Usability Issue	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Poolshark Labs's smart contracts.

In our audit, we sought to answer the following questions:

- Does the Poolshark protocol maintain all positions correctly, including when users mint multiple positions in overlapping ranges or when prices move between different ranges?
- Can a malicious user manipulate the value of another user's position?
- Can a malicious user game the system to steal funds from the protocol?
- Can a user always retrieve their funds after a mint by burning?
- Are mints allowed only when there are enough observations?
- Can the protocol be vulnerable to oracle manipulation?
- Are swaps calculated correctly? Do all swap transactions stay under slippage limits?
- Do the AMM math libraries function as expected?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following technique:

- *Fuzzing/Property-based Testing*. We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the (/cover/contracts) folder of the source code provided by the Poolshark Labs developers, which contains the smart contract implementation of the Poolshark Protocol.

Methodology. Veridise auditors inspected provided tests, and read the Poolshark Protocol documentation. They then began a manual audit of the code assisted by tooling. During the audit, the Veridise auditors regularly met with the Poolshark Labs developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ALL-PSH-001	Incorrect delta calculation on transfer	High	Fixed
V-ALL-PSH-002	Swap amount incorrectly calculated	High	Fixed
V-ALL-PSH-003	Incorrect delta calculation on delta-tick exchange	High	Fixed
V-ALL-PSH-004	Potential overflow on average tick calculation	Medium	Fixed
V-ALL-PSH-005	Vulnerability to oracle manipulation	Medium	Fixed
V-ALL-PSH-006	Stashed amount ignored in tick removal	Medium	Fixed
V-ALL-PSH-007	Liquidity not recalculated after partial mints	Medium	Intended
V-ALL-PSH-008	Bogus burn event	Medium	Fixed
V-ALL-PSH-009	Missing input validation in Positions.validate()	Medium	Invalid
V-ALL-PSH-010	Linked list manipulation	Medium	Fixed
V-ALL-PSH-011	Lack of validation on mint	Low	Acknowledged
V-ALL-PSH-012	Potentially unsafe typecast in Ticks.quote	Low	Fixed
V-ALL-PSH-013	No tick node deletion	Low	Fixed
V-ALL-PSH-014	Potential Denial of Service	Low	Fixed
V-ALL-PSH-015	No revert on cPL > 0	Warning	Fixed
V-ALL-PSH-016	Improvements to initialization of CoverPool	Info	Fixed
V-ALL-PSH-017	Unnecessary typecasts	Info	Fixed
V-ALL-PSH-018	Unimplemented ownership transfer in CoverPool	Info	Fixed
V-ALL-PSH-019	Unnecessary return values	Info	Fixed
V-ALL-PSH-020	Add option to burn percentage of position	Info	Fixed
V-ALL-PSH-021	Validate functions should not update state	Info	Open

4.1 Detailed Description of Bugs

4.1.1 V-ALL-PSH-001: Incorrect delta calculation on transfer

Severity	High	**Commit**	f8d337b
Type	Logic Error	**Status**	Fixed
Files	libraries/Deltas.sol		
Functions	transferMax		

In `transferMax`, the line `fromDeltas.amountOutDeltaMax = 0;` should instead set `fromDeltas.amountInDeltaMax`.

```
{
            uint128 amountInDeltaMaxChange = uint128(uint256(fromDeltas.
    amountInDeltaMax) * percentInTransfer / 1e38);
            if (fromDeltas.amountInDeltaMax > amountInDeltaMaxChange) {
                fromDeltas.amountInDeltaMax -= amountInDeltaMaxChange;
                toDeltas.amountInDeltaMax += amountInDeltaMaxChange;
            } else {
                toDeltas.amountInDeltaMax += fromDeltas.amountInDeltaMax;
                fromDeltas.amountOutDeltaMax = 0;
            }
        }
        {
            uint128 amountOutDeltaMaxChange = uint128(uint256(fromDeltas.
    amountOutDeltaMax) * percentOutTransfer / 1e38);
            if (fromDeltas.amountOutDeltaMax > amountOutDeltaMaxChange) {
                fromDeltas.amountOutDeltaMax -= amountOutDeltaMaxChange;
                toDeltas.amountOutDeltaMax   += amountOutDeltaMaxChange;
            } else {
                toDeltas.amountOutDeltaMax += fromDeltas.amountOutDeltaMax;
                fromDeltas.amountOutDeltaMax = 0;
            }
        }
```

Recommendation Change `fromDeltas.amountOutDeltaMax = 0;` to `fromDeltas.amountInDeltaMax = 0;`.

Developer Response Fixed in commit 79e2bb6.

4.1.2 V-ALL-PSH-002: Swap amount incorrectly calculated

Severity	High	Commit	f8d337b
Type	Logic Error	Status	Fixed
Files	Ticks.sol		
Functions	quote		

`Ticks.quote` is a key function for correctly calculating swap prices and the amount output, returning the `amountOut` given a specific `priceLimit` : the way this priceLimit is handled is as follows:

```
...
uint256 nextTickPrice = state.latestPrice;
uint256 nextPrice = nextTickPrice;

        // determine input boost from tick auction
        cache.auctionBoost = ((cache.auctionDepth <= state.auctionLength) ? cache.auctionDepth : state.auctionLength) * 1e14 / state.auctionLength * uint16(state.tickSpread);
        cache.inputBoosted = cache.input * (1e18 + cache.auctionBoost) / 1e18;

        if (zeroForOne) {
            // Trading token 0 (x) for token 1 (y).
            // price  is decreasing.
            if (priceLimit > nextPrice) {
                // stop at price limit
                nextPrice = priceLimit;
            }
...
```

This `nextPrice` , which takes into account the priceLimit, is used in calculating the `amountOut` : specifically, when there's supposed to be some remaining amount remaining in `cache.input`

```
uint256 maxDx = DyDxMath.getDx(cache.liquidity, nextPrice, cache.price, false);
            // check if we can increase input to account for auction
            // if we can't, subtract amount inputted at the end
            // store amountInDelta in pool either way
            // putting in less either way
            if (cache.inputBoosted <= maxDx) {
                // We can swap within the current range.
                uint256 liquidityPadded = cache.liquidity << 96;
                // calculate price after swap
                uint256 newPrice = FullPrecisionMath.mulDivRoundingUp(
                    liquidityPadded,
                    cache.price,
                    liquidityPadded + cache.price * cache.inputBoosted
                );
                /// @auditor - check tests to see if we need overflow handle
                // if (!(nextTickPrice <= newPrice && newPrice < cache.price)) {
                //     console.log('overflow check');
                //     newPrice = uint160(FullPrecisionMath.divRoundingUp(liquidityPadded, liquidityPadded / cache.price + cache.input));
                // }
```

```
20                 amountOut = DyDxMath.getDy(cache.liquidity, newPrice, cache.price,
    false);
21                 cache.price = uint160(newPrice);
22                 cache.amountInDelta = maxDx - maxDx * cache.input / cache.
    inputBoosted;
23                 cache.input = 0;
24             } else if (maxDx > 0) {
25                 amountOut = DyDxMath.getDy(cache.liquidity, nextPrice, cache.price,
    false);
26                 cache.price = nextPrice;
27                 cache.amountInDelta = maxDx - maxDx * cache.input / cache.
    inputBoosted;
28                 cache.input -= maxDx * cache.input / cache.inputBoosted; /// @dev -
    convert back to input amount
29             }
30         } else {
31             // Price is increasing.
32             if (priceLimit < nextPrice) {
33                 // stop at price limit
34                 nextPrice = priceLimit;
35             }
36             uint256 maxDy = DyDxMath.getDy(cache.liquidity, cache.price, nextPrice,
    false);
37             if (cache.inputBoosted <= maxDy) {
38                 // We can swap within the current range.
39                 // Calculate new price after swap:  P  =  y /L.
40                 uint256 newPrice = cache.price +
41                     FullPrecisionMath.mulDiv(cache.inputBoosted, Q96, cache.liquidity
    );
42                 // Calculate output of swap
43                 amountOut = DyDxMath.getDx(cache.liquidity, cache.price, newPrice,
    false);
44                 cache.price = newPrice;
45                 cache.amountInDelta = cache.inputBoosted - cache.input;
46                 cache.input = 0;
47             } else if (maxDy > 0) {
48                 amountOut = DyDxMath.getDx(cache.liquidity, cache.price,
    nextTickPrice, false);
49                 cache.price = nextPrice;
50                 cache.amountInDelta = maxDy - maxDy * cache.input / cache.
    inputBoosted;
51                 cache.input -= maxDy * cache.input / cache.inputBoosted + 1; /// @dev
     - handles rounding errors with amountInDelta
52             }
53         }
```

The calculation of `amountOut` in the `else if (maxDy > 0)` clause does not use the intended `nextPrice`, but instead uses `nextTickPrice` which doesn't account for the `priceLimit` previously indicated.

Impact This calculation ignores the desired price limit of the user, meaning more slippage than intended could impact the user.

Recommendation Use `nextPrice` instead of `nextTickPrice`.

Developer Response Fixed in commit e633e42.

4.1.3 V-ALL-PSH-003: Incorrect delta calculation on delta-tick transfer

Severity	High	**Commit**	f8d337b
Type	Logic Error	**Status**	Fixed
Files	Deltas.sol		
Functions	to		

`to` transfers delta-amounts from a delta to a tick. Here, however, it transfers the delta amount incorrectly - `toTick.deltas.amountOutDelta` should increase by only `fromDeltas.amountOutDelta`, not the max.

```
function to(
        ICoverPoolStructs.Deltas memory fromDeltas,
        ICoverPoolStructs.Tick memory toTick
    ) external pure returns (
        ICoverPoolStructs.Deltas memory,
        ICoverPoolStructs.Tick memory
    ) {
        toTick.deltas.amountInDelta     += fromDeltas.amountInDelta;
        toTick.deltas.amountInDeltaMax  += fromDeltas.amountInDeltaMax;
        toTick.deltas.amountOutDelta    += fromDeltas.amountOutDeltaMax;
        toTick.deltas.amountOutDeltaMax += fromDeltas.amountOutDeltaMax;
        fromDeltas = ICoverPoolStructs.Deltas(0,0,0,0);
        return (fromDeltas, toTick);
    }
```

Recommendation Change from `toTick.deltas.amountOutDelta += fromDeltas.amountOutDeltaMax` to `toTick.deltas.amountOutDelta += fromDeltas.amountOutDelta;`.

Developer Response Fixed in commit 7632bec.

4.1.4 V-ALL-PSH-004: Potential overflow on average tick calculation

Severity	Medium	**Commit**	f8d337b
Type	Data Validation	**Status**	Fixed
Files	TwapOracle.sol		
Functions	calculateAverageTick		

Currently, the TWAP oracle calculates average tick based on the deployed chain's blocktime and an input `uint16 twapLength`. The result is stored in a `uint32[] secondsAgos`: however, there is a cast to `int32` that may potentially overflow

```
...
averageTick = int24(((tickCumulatives[0] - tickCumulatives[1]) / (int32(secondsAgos
    [1]))));
...
```

Impact If the typecast overflows (which it can if `twapLength * blocktime > type(int32).max`), it may pass the checks below which only checks for strict equality to `TickMath.MAX_TICK` and `TickMath.MIN_TICK`. An inaccurate average tick calculation directly affects `syncLatest`, which is performed before the execution of any of the major functions in `CoverPool`

Recommendation Place limits on the values of `twapLength` and `blocktime` to ensure no overflow.

Developer Response Fixed in commit db9e57e.

4.1.5 V-ALL-PSH-005: Vulnerability to oracle manipulation

Severity	Medium	**Commit**	f8d337b
Type	Logic Error	**Status**	Fixed
Files	TwapOracle.sol		
Functions	N/A		

The protocol relies on a price oracle to calculate the TWAP based on prices determined from the underlying range pool. This appears to be the only source of prices, meaning that any attack which compromises the prices reported by this oracle could severely manipulate the behavior of the pool. As an an example, if the oracle is comprised, an incorrect change in prices could force the cover pool to auction off liquidity when it should not.

Impact A property functioning TWAP oracle is imperative for the correct operation of the protocol. Without a correctly functioning oracle, LP providers cannot appropriately hedge as intended.

Recommendation To reduce the risk, it is suggested that multiple price oracles are queried and averaged so that there is not a single point of failure.

Developer Response This issue is fixed by rate-limiting the price move as a function of `auctionLength` and `tickSpread` in `Epochs.syncLatest()`.

4.1.6 V-ALL-PSH-006: Stashed amount ignored in tick removal

Severity	Medium	**Commit**	f8d337b
Type	Logic Error	**Status**	Fixed
Files	Ticks.sol		
Functions	remove		

`amountStashed` is unused in `Ticks.remove()`. If amount stashed should be considered during tick removal, this logic should be added. However, we suspect the argument should just be removed from the function.

```
function remove(
        mapping(int24 => ICoverPoolStructs.Tick) storage ticks,
        mapping(int24 => ICoverPoolStructs.TickNode) storage tickNodes,
        ICoverPoolStructs.GlobalState memory state,
        int24 lower,
        int24 upper,
        uint128 amount,
        uint128 amountStashed,
        bool isPool0,
        bool removeLower,
        bool removeUpper
    ) external {
        {
            ICoverPoolStructs.Tick memory tickLower = ticks[lower];
            if (removeLower) {
                if (isPool0) {
                    tickLower.liquidityDelta += int128(amount);
                    tickLower.liquidityDeltaMinus -= amount;
                } else {
                    tickLower.liquidityDelta -= int128(amount);
                }
            }
            /// @dev - not deleting ticks just yet
            ticks[lower] = tickLower;
        }

        {
            ICoverPoolStructs.Tick memory tickUpper = ticks[upper];
            if (removeUpper) {
                if (isPool0) {
                    tickUpper.liquidityDelta -= int128(amount);
                } else {
                    tickUpper.liquidityDelta += int128(amount);
                    tickUpper.liquidityDeltaMinus -= amount;
                }
            }
            ticks[upper] = tickUpper;
        }
    }
```

Recommendation Remove the `amountStashed` argument to `Ticks.remove`.

Developer Response Fixed in commit 00dc9dd. Tick deletion added in commit 97047ff.

4.1.7 V-ALL-PSH-007: Liquidity not recalculated after partial mints

Severity	Medium	**Commit**	f8d337b
Type	Usability Issue	**Status**	Intended
Files	Positions.sol		
Functions	validate		

Inside of `Positions.validate`, the following logic handles cases of partial mints by setting `priceUpper` and `priceLower` to their updated versions respectively. These parameter updates, however, aren't used since the actual calculation of `liquidityMinted` takes place before these partial mint scenarios are handled.

```
liquidityMinted = DyDxMath.getLiquidityForAmounts(
            priceLower,
            priceUpper,
            params.zeroForOne ? priceLower : priceUpper,
            params.zeroForOne ? 0 : uint256(params.amount),
            params.zeroForOne ? uint256(params.amount) : 0
        );

// handle partial mints
        if (params.zeroForOne) {
            if (params.upper >= params.state.latestTick) {
                params.upper = params.state.latestTick - int24(params.state.
    tickSpread);
                params.upperOld = params.state.latestTick;
                uint256 priceNewUpper = TickMath.getSqrtRatioAtTick(params.upper);
                params.amount -= uint128(
                    DyDxMath.getDx(liquidityMinted, priceNewUpper, priceUpper, false)
                );
                priceUpper = priceNewUpper;
            }
        } else {
            if (params.lower <= params.state.latestTick) {
                params.lower = params.state.latestTick + int24(params.state.
    tickSpread);
                params.lowerOld = params.state.latestTick;
                uint256 priceNewLower = TickMath.getSqrtRatioAtTick(params.lower);
                params.amount -= uint128(
                    DyDxMath.getDy(liquidityMinted, priceLower, priceNewLower, false)
                );
                priceLower = priceNewLower;
            }
        }
```

Impact Due to the incorrect timing of the calculation, partial mint cases are not actually taken into account. This may lead to sometimes incorrect calculations of liquidity positions.

Recommendation Move the calculations down after the if-else block.

Developer Response This is intended behavior as `params.amount` is adjusted appropriately. Based on this, auditors suggested removing the unecessary writes to `priceLower` and `priceUpper`, however, developers stated that although these updated values are not used in this version of the code, in a later commit they use these values so they will keep them in the code.

4.1.8 V-ALL-PSH-008: Bogus burn event

Severity	Medium	**Commit**	f8d337b
Type	Data Validation	**Status**	Fixed
Files	CoverPool.sol		
Functions	burn		

In CoverPool, core functions such as `mint` use `Positions.validate` to verify input parameters.

```
function mint(
        int24 lowerOld,
        int24 lower,
        int24 claim,
        int24 upper,
        int24 upperOld,
        uint128 amountDesired,
        bool zeroForOne
    ) external lock {
        ...
        (lowerOld, lower, upper, upperOld, amountDesired, liquidityMinted) =
    Positions.validate(
            ValidateParams(lowerOld, lower, upper, upperOld, zeroForOne,
    amountDesired, globalState)
        );
```

Similar input validation should be present in `burn`; however, it is not as shown.

```
function burn(
        int24 lower,
        int24 claim,
        int24 upper,
        bool zeroForOne,
        uint128 amount
    ) external lock {
        GlobalState memory state = globalState;
        if (block.number != state.lastBlock) {
            (state, pool0, pool1) = Epochs.syncLatest(
                ticks0,
                ticks1,
                tickNodes,
                pool0,
                pool1,
                state
            );
        }
        //TODO: burning liquidity should take liquidity out past the current auction

        // Ensure no overflow happens when we cast from uint128 to int128.
        if (amount > uint128(type(int128).max)) revert LiquidityOverflow();

        if (claim != (zeroForOne ? upper : lower) || claim == state.latestTick) {
            // update position and get new lower and upper
            state = Positions.update(
```

```
                zeroForOne ? positions0 : positions1,
                zeroForOne ? ticks0 : ticks1,
                tickNodes,
                state,
                zeroForOne ? pool0 : pool1,
                UpdateParams(msg.sender, lower, upper, claim, zeroForOne, amount)
            );
        }
        //TODO: add PositionUpdated event
        // if position hasn't changed remove liquidity
        else {
            (, state) = Positions.remove(
                zeroForOne ? positions0 : positions1,
                zeroForOne ? ticks0 : ticks1,
                tickNodes,
                state,
                RemoveParams(msg.sender, lower, upper, zeroForOne, amount)
            );
        }
        //TODO: get token amounts from _updatePosition return values
        //TODO: need to know old ticks and new ticks
        emit Burn(msg.sender, lower, upper, claim, zeroForOne, amount);
        globalState = state;
    }
```

For many invalid inputs, the call to `burn` will essentially be a no-op. However, because there is an event, a malicious user could use this to emit a bogus event indicating a burn completed that was not really valid.

Impact This could be used to manipulate the event log for the protocol, which could cause issues with external applications relying on that event log.

Recommendation Add in parameter validation to ensure all invalid burns revert.

Developer Response Fixed in commit c994b53.

4.1.9 V-ALL-PSH-009: Missing input validation in Positions.validate()

Severity	Medium	**Commit**	f8d337b
Type	Logic Error	**Status**	Invalid
Files	Positions.sol		
Functions	validate		

Inside of `Positions.update`, there is a comment regarding an invariant of the protocol:

```
// @auditor - user cannot add liquidity if auction is active; checked for in
    Positions.validate()
```

The property is supposed to be checked for in `Positions.validate`; however, the check for this property is not present

```
function validate(ICoverPoolStructs.ValidateParams memory params)
        external
        pure
        returns (
            int24,
            int24,
            int24,
            int24,
            uint128,
            uint256 liquidityMinted
        )
    {
        if (params.lower < TickMath.MIN_TICK) revert InvalidLowerTick();
        if (params.upper > TickMath.MAX_TICK) revert InvalidUpperTick();
        if (params.lower % int24(params.state.tickSpread) != 0) revert
    InvalidLowerTick();
        if (params.upper % int24(params.state.tickSpread) != 0) revert
    InvalidUpperTick();
        if (params.amount == 0) revert InvalidPositionAmount();
        if (params.lower >= params.upper || params.lowerOld >= params.upperOld)
            revert InvalidPositionBoundsOrder();
        if (params.zeroForOne) {
            if (params.lower >= params.state.latestTick) revert
    InvalidPositionBoundsTwap();
        } else {
            if (params.upper <= params.state.latestTick) revert
    InvalidPositionBoundsTwap();
        }
        uint256 priceLower = uint256(TickMath.getSqrtRatioAtTick(params.lower));
        uint256 priceUpper = uint256(TickMath.getSqrtRatioAtTick(params.upper));

        liquidityMinted = DyDxMath.getLiquidityForAmounts(
            priceLower,
            priceUpper,
            params.zeroForOne ? priceLower : priceUpper,
            params.zeroForOne ? 0 : uint256(params.amount),
            params.zeroForOne ? uint256(params.amount) : 0
        );
```

```
35
36          // handle partial mints
37          if (params.zeroForOne) {
38              if (params.upper >= params.state.latestTick) {
39                  params.upper = params.state.latestTick - int24(params.state.
    tickSpread);
40                  params.upperOld = params.state.latestTick;
41                  uint256 priceNewUpper = TickMath.getSqrtRatioAtTick(params.upper);
42                  params.amount -= uint128(
43                      DyDxMath.getDx(liquidityMinted, priceNewUpper, priceUpper, false)
44                  );
45                  priceUpper = priceNewUpper;
46              }
47          } else {
48              if (params.lower <= params.state.latestTick) {
49                  params.lower = params.state.latestTick + int24(params.state.
    tickSpread);
50                  params.lowerOld = params.state.latestTick;
51                  uint256 priceNewLower = TickMath.getSqrtRatioAtTick(params.lower);
52                  params.amount -= uint128(
53                      DyDxMath.getDy(liquidityMinted, priceLower, priceNewLower, false)
54                  );
55                  priceLower = priceNewLower;
56              }
57          }
58
59          if (liquidityMinted > uint128(type(int128).max)) revert LiquidityOverflow();
60          if (params.lower == params.upper) revert InvalidPositionBoundsTwap();
61
62          return (
63              params.lowerOld,
64              params.lower,
65              params.upper,
66              params.upperOld,
67              params.amount,
68              liquidityMinted
69          );
70      }
```

Developer Response This check is handled indirectly by the bounds checks for valid positions. The relevant checks can be found in `Positions.add`. For clarity, auditors suggested making these checks more explicit and well-documented for future maintainability.

4.1.10 V-ALL-PSH-010: Linked list manipulation

Severity	Medium	**Commit**	f8d337b
Type	Logic Error	**Status**	Fixed
Files	CoverPool.sol		
Functions	mint		

The `mint` function currently has the following interface:

```
function mint(
        int24 lowerOld,
        int24 lower,
        int24 claim,
        int24 upper,
        int24 upperOld,
        uint128 amountDesired,
        bool zeroForOne
    ) external
```

Two of the arguments `lowerOld` and `upperOld` are used to appropriately add the position in an internal linked list maintaining positions. Some illegal settings of these values (such as having `lowerOld >= upperOld` are pruned, however, it is still possible that a malicious could use these values to mess with the internal `tickNodes` data structure.

Recommendation Calculate or store these values rather than rely on the external user providing them.

Developer Response Fixed in commit 8163ea1. To fix this issue, developers introduced a Tick bitmap to avoid linked list manipulation/breakage. Furthermore, they do not rely on the user to provide inputs to maintain the bitmap.

4.1.11 V-ALL-PSH-011: Lack of validation on mint

Severity	Low	Commit	f8d337b
Type	Data Validation	Status	Acknowledged
Files	CoverPool.sol		
Functions	mint		

The comments on mint indicate that mint should only be called from the CL pool manager contract. No caller validation, however, is made anywhere inside the function.

```
/// @dev Mints LP tokens - should be called via the CL pool manager contract.
    function mint(
        int24 lowerOld,
        int24 lower,
        int24 claim,
        int24 upper,
        int24 upperOld,
        uint128 amountDesired,
        bool zeroForOne
    ) external lock {
```

Recommendation Add input validation as suggested.

Developer Response Developer's suggested this comment is no longer relevant, i.e., no caller validation is required.

4.1.12 V-ALL-PSH-012: Potentially unsafe typecast in Ticks.quote

Severity	Low	**Commit**	f8d337b
Type	Data Validation	**Status**	Fixed
Files	Ticks.sol		
Functions	quote		

The following implementation of quote contains a typecast `uint160(newPrice)` that may be unsafe because `newPrice` is a `uint256`.

```
if (cache.inputBoosted <= maxDx) {
                // We can swap within the current range.
                uint256 liquidityPadded = cache.liquidity << 96;
                // calculate price after swap
                uint256 newPrice = FullPrecisionMath.mulDivRoundingUp(
                    liquidityPadded,
                    cache.price,
                    liquidityPadded + cache.price * cache.inputBoosted
                );
                /// @auditor - check tests to see if we need overflow handle
                // if (!(nextTickPrice <= newPrice && newPrice < cache.price)) {
                //     console.log('overflow check');
                //     newPrice = uint160(FullPrecisionMath.divRoundingUp(
    liquidityPadded, liquidityPadded / cache.price + cache.input));
                // }
                amountOut = DyDxMath.getDy(cache.liquidity, newPrice, cache.price,
    false);
                cache.price = uint160(newPrice);
                cache.amountInDelta = maxDx - maxDx * cache.input / cache.
    inputBoosted;
                cache.input = 0;
            } else if (maxDx > 0) {
                amountOut = DyDxMath.getDy(cache.liquidity, nextPrice, cache.price,
    false);
                cache.price = nextPrice;
                cache.amountInDelta = maxDx - maxDx * cache.input / cache.
    inputBoosted;
                cache.input -= maxDx * cache.input / cache.inputBoosted; /// @dev -
    convert back to input amount
            }
```

Impact If prices were such that this caused an overflow, it is possible the price would be unable to update and the protocol could get stuck.

Recommendation To avoid any possibility of overflow, we recommend removing the typecast as `cache.price` is also `uint256`.

Developer Response Fixed in commit 545e2d2.

4.1.13 V-ALL-PSH-013: No tick node deletion

Severity	Low	**Commit**	f8d337b
Type	Maintainability	**Status**	Fixed
Files	Ticks.sol		
Functions	remove		

In `remove` there seems to be code regarding deleting various ticks that are intended to be included, but is either not yet implemented or obsolete.

```
// if (deleteLowerTick) {
        //     // Delete lower tick.
        //     int24 previous = tickNodes[lower].previousTick;
        //     int24 next     = tickNodes[lower].nextTick;
        //     if(next != upper || !deleteUpperTick) {
        //         tickNodes[previous].nextTick = next;
        //         tickNodes[next].previousTick = previous;
        //     } else {
        //         int24 upperNextTick = tickNodes[upper].nextTick;
        //         tickNodes[tickNodes[lower].previousTick].nextTick = upperNextTick;
        //         tickNodes[upperNextTick].previousTick = previous;
        //     }
        // }
        // if (deleteUpperTick) {
        //     // Delete upper tick.
        //     int24 previous = tickNodes[upper].previousTick;
        //     int24 next     = tickNodes[upper].nextTick;

        //     if(previous != lower || !deleteLowerTick) {
        //         tickNodes[previous].nextTick = next;
        //         tickNodes[next].previousTick = previous;
        //     } else {
        //         int24 lowerPrevTick = tickNodes[lower].previousTick;
        //         tickNodes[lowerPrevTick].nextTick = next;
        //         tickNodes[next].previousTick = lowerPrevTick;
        //     }
        // }
        /// @dev - we can never delete ticks due to amount deltas
```

Without deleting ticks, it is possible for a malicious user to add a significant number of nodes to the `tickNodes` linked list, causing updates to be slower and less gas efficient.

Impact This could allow a malicious user to increase the cost of syncing.

Recommendation Remove tick nodes on removal to improve update efficiency.

Developer Response Fixed in commit c9e3981.

4.1.14 V-ALL-PSH-014: Potential Denial of Service

Severity	Low	**Commit**	f8d337b
Type	Denial of Service	**Status**	Fixed
Files	Epochs.sol		
Functions	syncLatest		

The function `syncLatest` relies on multiple `while(true)` loops which iterate through ticks to perform updates. Below is an example:

```
while (true) {
            // rollover deltas from current auction
            (cache, pool0) = _rollover(cache, pool0, true);
            // accumulate to next tick
            ICoverPoolStructs.AccumulateOutputs memory outputs;
            outputs = _accumulate(
                tickNodes[cache.nextTickToAccum0],
                tickNodes[cache.nextTickToCross0],
                ticks0[cache.nextTickToCross0],
                ticks0[cache.nextTickToAccum0],
                cache.deltas0,
                state.accumEpoch,
                true,
                nextLatestTick > state.latestTick
                    ? cache.nextTickToAccum0 < cache.stopTick0
                    : cache.nextTickToAccum0 > cache.stopTick0
            );
            cache.deltas0 = outputs.deltas;
            tickNodes[cache.nextTickToAccum0] = outputs.accumTickNode;
            tickNodes[cache.nextTickToCross0] = outputs.crossTickNode;
            ticks0[cache.nextTickToCross0] = outputs.crossTick;
            ticks0[cache.nextTickToAccum0] = outputs.accumTick;
            //cross otherwise break
            if (cache.nextTickToAccum0 > cache.stopTick0) {
                (pool0.liquidity, cache.nextTickToCross0, cache.nextTickToAccum0) =
    _cross(
                    tickNodes[cache.nextTickToAccum0],
                    ticks0[cache.nextTickToAccum0].liquidityDelta,
                    cache.nextTickToCross0,
                    cache.nextTickToAccum0,
                    pool0.liquidity,
                    true
                );
                if (cache.nextTickToCross0 == cache.nextTickToAccum0) {
                    revert InfiniteTickLoop0(cache.nextTickToAccum0);
                }
            } else break;
        }
```

`syncLatest` is called by almost every public facing function in the protocol — thus if these loops are infinite or consume a significant amount of gas, almost every function of the protocol could be impacted.

Recommendation We suggest converting the logic to enable stronger guarantees that looping will terminate in a reasonable amount of time. For instance, if ticks were stored in a map, looping over the length of the map should be sufficient to guarantee termination. Another possible approach is to cap iterations of the loop based on the maximum number of possible ticks between the current prices.

Developer Resposne The developer fix is in commit f2319fa. This commit adds an `params.sync` argument to `burn` which allows a user to skip the sync when desired. This does not necessarily totally prevent a DoS attack, but provides the user a mechanism to withdraw their funds even in the case of a DoS attack.

4.1.15 V-ALL-PSH-015: No revert on cPL > 0

Severity	Warning	**Commit**	f8d337b
Type	Data Validation	**Status**	Fixed
Files	Positions.sol		
Functions	add		

As per comments, `add` should revert if `cPL > 0`, but there are no checks to ensure that will happen.

```
1 //TODO: if cPL is > 0, revert
```

Recommendation Add the checks as commented.

Developer Response Fixed in commit f66ed4e.

4.1.16 V-ALL-PSH-016: Improvements to initialization of CoverPool

Severity	Info	Commit	f8d337b
Type	Data Validation	Status	Fixed
Files	CoverPool.sol		
Functions	constructor		

The constructor of CoverPool first initializes an empty state, then proceeds to set state as follows:

```
// set global state
GlobalState memory state = GlobalState(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, IRangePool(
    address(0)));
state.tickSpread    = _tickSpread;
state.twapLength    = _twapLength;
state.auctionLength = _auctionLength;
state.genesisBlock  = uint32(block.number);
state.inputPool     = IRangePool(_inputPool);
```

`lastBlock` and `auctionStart` remain unset, and the explicit write to 0 state is unnecessary since Solidity sets variables to 0 by default.

Recommendation Set the input parameters in the same write to `GlobalState memory state` and set `lastBlock` and `auctionStart` to the same block as `genesisBlock` (otherwise they get left as 0).

Developer Response Fixed in commit b5d7f65.

4.1.17 V-ALL-PSH-017: Unnecessary typecasts

Severity	Info	**Commit**	f8d337b
Type	Logic Error	**Status**	Fixed
Files	CoverPoolFactory.sol, Positions.sol		
Functions	createCoverPool		

In the following event, `tickSpread` is already `int16`, rendering a cast to `int26` to be unnecessary.

```
// emit event for indexers
emit PoolCreated(token0, token1, uint24(feeTier), int16(tickSpread), twapLength,
    auctionLength, pool);
```

Similarly, in `Positions.add` the casts of `params.amount` to `uint128` are unnecessary as this value is already a `uint128`

Recommendation Removing the unnecessary typecasts.

Developer Response Fixed in commit b5d7f65.

4.1.18 V-ALL-PSH-018: Unimplemented ownership transfer

Severity	Info	**Commit**	f8d337b
Type	Maintainability	**Status**	Fixed
Files	CoverPool		
Functions	n/a		

Per comments, there is currently no implementation of a function to transfer ownership in CoverPool.

```
1 //TODO: create transfer function to transfer ownership
```

Recommendation Create the function as described.

Developer Response Fixed in commit 061ee49.

4.1.19 V-ALL-PSH-019: Unnecessary Return Values

Severity	Info	**Commit**	f8d337b
Type	Maintainability	**Status**	Fixed
Files	Ticks.sol, Positions.sol		
Functions	n/a		

`Ticks.insert` and `Positions.add` both return the state as if it is modified even though neither appears to actually modify the state.

Recommendation Adjust the functions such that they no longer return the state, since the return is nowhere used.

Developer Response Fixed in commit 7b6812b.

4.1.20 V-ALL-PSH-020: Add option to burn percentage of position

Severity	Info	**Commit**	f8d337b
Type	Maintainability	**Status**	Fixed
Files	CoverPool.sol		
Functions	n/a		

Currently, the burn function requires an LP to provide a "liquidity amount" to be burned. However, calculating such an amount is somewhat non-intuitive.

Recommendation Add an addition function which supports burning a percentage of a position.

Developer Response The developers introduced this feature in commit 30effa2. As of the time of writing, this commit was not yet merged but will soon be merged according to developers.

4.1.21 V-ALL-PSH-021: Validate functions should not update state

Severity	Info	**Commit**	f8d337b
Type	Maintainability	**Status**	Open
Files	Positions.sol		
Functions	n/a		

While reviewing the code, auditors were confused by the convention that "validation" functions were often used not only to perform validation but also update state. For instance, `Positions.validate` will not only validate a position, but can update the range to a valid range.

Recommendation We suggest separating any validation logic from any logic that update state to make the distinction clear.