Panoptic monorepo

Security Review

Cantina Managed review by:
R0bert, Lead Security Researcher
Phaze, Security Researcher

November 20, 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

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

1.3 Risk assessment

Severity	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

Panoptic is the perpetual, oracle-free options protocol built on the Ethereum blockchain.

From Oct 31st to Nov 9th the Cantina team conducted a review of panoptic-monorepo on commit hash 81b25f46. The team identified a total of **4** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 3
- Gas Optimizations: 0
- Informational: 0

3 Findings

3.1 Medium Risk

3.1.1 Wrong implementation of the expandEnforcedTickRange() function

Severity: Medium Risk

Context: contracts/SemiFungiblePositionManager.sol#L449

Description: The `SemiFungiblePositionManager` contract implements the function `expandEnforcedTickRange()` that recomputes and decreases `minEnforcedTick` and/or increases `maxEnforcedTick` for a given `poolId` if the following conditions are met:

- The token supply for one of the tokens was greater than `MIN_ENFORCED_TICKFILL_COST` at the last `initializeAMMPool` or `expandEnforcedTickRangeForPool` call for `poolId`.
- The token supply for one of the tokens meeting the first condition has decreased significantly since the last call.

During a `initializeAMMPool()` call the total supply of the `currency1` is used to compute the `minEnforcedTick` and the total supply of the `currency0` is used to compute the `maxEnforcedTick`. However, `expandEnforcedTickRange()` wrongly uses the `currency1` total supply to recalculate `maxEnforcedTick`.

On the other hand, within the `expandEnforcedTickRange()` function, in the `minEnforcedTick` calculation the `getApproxTickWithMaxAmount` result should be negated while for the `maxEnforcedTick` it should not.

This implementation would lead to an enforced tick range that doesn't accurately reflect the true liquidity or price dynamics of the pool and therefore make the liquidity addition DoS attack economically viable and easy to execute.

Recommendation: Update the `expandEnforcedTickRange()` so it uses the `currency0` to compute the `maxEnforcedTick`. Finally, correct the `getApproxTickWithMaxAmount` result negating it for the `minEnforcedTick` calculation and removing the negation for the `maxEnforcedTick` result.

Panoptic: Fixed in commit 7463cea3.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Inability to narrow the enforced tick range in the expandEnforcedTickRange function

Severity: Low Risk

Context: contracts/SemiFungiblePositionManager.sol#L426, contracts/SemiFungiblePositionManager.sol#L422

Description: The `expandEnforcedTickRange()` function in the `SemiFungiblePositionManager` contract is designed to adjust the enforced tick range for a given Uniswap V4 pool based on the tokens' total supply. However, the function only allows for expanding the tick range and can not narrow it. This limitation could be a problem when:

- Dealing with inflationary tokens/tokens with an increasing total supply: Where the enforced tick range should ideally be narrowed to reflect the new token supply dynamics.
- Burnable/bridgeable tokens: Tokens like USDC can be burned or bridged to another chain, temporarily reducing their total supply on the original chain. A malicious actor could exploit this by bridging a large amount of the token, reducing the total supply, and then calling `expandEnforcedTickRange` to expand the enforced tick range. Once the enforced tick range has been expanded, the actor can bridge the tokens back which would restore the initial total supply. However, because the function only allows expanding the tick range and not narrowing it, the excessively wide enforced tick range remains.

The inability to narrow the tick range could lead to an excessively wide enforced tick range that would not reflect the true liquidity or price dynamics of the pool and therefore make the liquidity addition DoS attack economically viable.

Recommendation: Instead of calculating the `minEnforcedTick` and `maxEnforcedTick` once during the pool initialization and only when the `expandEnforcedTickRange` function is called, consider executing the calculation directly in the `_createPositionInAMM()` internal function. Moreover, its also recommended in this case, to also allow for the narrowing of the tick range. As now, the calculation would be implemented directly in the `_createPositionInAMM()` function, there is no room for flashloan/front-running to deny the creation of a position.

Panoptic: Acknowledged. We are staying with the current approach in order to reduce gas costs -- the lower bound of `MIN_ENFORCED_TICKFILL_COST` covers the vast majority of bridgeable/wrapped tokens economically (given the example of USDC, `MIN_ENFORCED_TICKFILL_COST` is over 50,000 times greater than the current total supply of that token, which itself already represents 30 billion dollars in value). The supply-based tick calculations are more relevant for very-high-supply tokens, which tend to have either a fixed or monotonically decreasing supply. In the rare circumstance where a tick range for a pool is set too wide to prevent liquidity addition DoS, that pool will be automatically blacklisted from our interface.

Canitna Managed: Acknowledged.

3.2.2 Unsafe type casting operations and undocumented assumptions

Severity: Low Risk

Context: contracts/libraries/PanopticMath.sol#L914-L916, contracts/libraries/PanopticMath.sol#L964-L975, contracts/SemiFungiblePositionManager.sol#L1249-L1257

Description: The codebase contains several instances of unchecked type casting operations and undocumented assumptions about value ranges that could lead to vulnerabilities in the protocol. These issues make the code harder to audit, maintain, and could potentially hide subtle bugs.

Some examples can be seen in:

1. Unchecked casting of `protocolLoss1 + collateralDelta1` to `int128` in `PanopticMath.sol`:

```
haircutBase = LeftRightSigned.wrap(longPremium.rightSlot()).toLeftSlot(
int128(protocolLoss1 + collateralDelta1)
);
```

2. Potentially unsafe conversion between `LeftRightSigned` and `LeftRightUnsigned` types and undocumented assumption about sign assumptions in `movedInLeg` in `SemiFungiblePositionManager.sol`:

```
amountToCollect = LeftRightUnsigned.wrap(
    uint256(
        LeftRightSigned.unwrap(
            LeftRightSigned.wrap(int256(LeftRightUnsigned.unwrap(amountToCollect))).sub(
                movedInLeg
            )
        )
    )
);
```

3. A series of nested casting operations are performed without documenting assumptions about the sign and range of `_premiasByLeg` in `PanopticMath.sol`:

```
LeftRightSigned haircutAmounts = LeftRightSigned
    .wrap(
        int128(
            uint128(
                Math.unsafeDivRoundingUp(
                    uint128(-_premiasByLeg[i][leg].rightSlot()) *
                        uint256(uint128(haircutBase.rightSlot())),
                    uint128(longPremium.rightSlot())
                )
            )
        )
    )
```

In these examples, there are no explicit checks to ensure the values being cast will fit within their target types, nor documentation explaining the assumed valid ranges for these operations.

Recommendation: Consider implementing safe casting utilities and explicit validation checks, e.g. by using OpenZeppelin's `SafeCast` library and creating safe casting libraries for the custom `LeftRight` types.

Further, document all assumptions about value ranges in comments, e.g:

```
// protocolLoss1 + collateralDelta1 is guaranteed to fit in int128 because...
haircutBase = LeftRightSigned.wrap(longPremium.rightSlot()).toLeftSlot(
   safeToInt128(protocolLoss1 + collateralDelta1)
);
```

Panoptic: Acknowledged and partially fixed. Comments explaining the underlying assumptions have been left in the following commits:

- V1: commit dab4a5bb.
- V1.1: 58fe604.

However, no explicit checks or further validation were applied.

Cantino Managed: Acknowledged.

3.2.3 Potential reentrancy vulnerability in `settleLiquidation`

Severity: Low Risk

Context: contracts/CollateralTracker.sol#L927-L1018, contracts/libraries/SafeTransferLib.sol#L16-L28

Description: In the CollateralTracker contract, the `settleLiquidation()` function contains a potential reentrancy vulnerability when refunding ETH to the liquidator. The issue occurs because the function performs an external ETH transfer before updating the liquidatee's balance state:

```
// ETH refund happens before state updates
if (msg.value > 0) SafeTransferLib.safeTransferETH(liquidator, msg.value);

uint256 liquidateeBalance = balanceOf[liquidatee];

if (type(uint248).max > liquidateeBalance) {
    unchecked {
        totalSupply += type(uint248).max - liquidateeBalance;
    }
    liquidateeBalance = 0;
} else {
    unchecked {
        liquidateeBalance -= type(uint248).max;
    }
}
```

A malicious liquidator could reenter the `settleLiquidation()` function, potentially manipulating the protocol's state before the original call completes its state updates.

Recommendation: Consider implementing the Checks-Effects-Interactions pattern by moving the ETH refund after all state changes:

```
function settleLiquidation(
    address liquidator,
    address liquidatee,
    int256 bonus
) external payable onlyPanopticPool {
    if (bonus < 0) {
        // ... existing logic ...
    } else {
        uint256 refundAmount = msg.value; // Store refund amount

        // State updates first
        uint256 liquidateeBalance = balanceOf[liquidatee];
        // ... rest of state update logic ...

        // ETH refund after all state changes
        if (refundAmount > 0) SafeTransferLib.safeTransferETH(liquidator, refundAmount);
    }
}
```

Alternatively, consider adding a nonReentrant modifier to the function if one is not already present in the broader context.

Panoptic: Fixed in commit 58fe6046.

Cantina Managed: Fix verified.

4 Appendix

4.1 Scope & Analysis

During the seven day engagement, Cantina Managed researchers conducted a comprehensive analysis of the codebase, focusing on the following Pull Requests (PRs) and commits:

Category	Identifier
Pull Requests	PR 143
	PR 144
	PR 148
Version 1 (Uniswap V3)	Commit `9b5eb397`
	Commit `723aa559`
	Commit `dab4a5bb`
Version 1.1 (Uniswap V4)	Commit `dc6ad9d4`
	Commit `7463cea3`
	Commit `5dabb49a`
	Commit `58fe6046`

4.1.1 PR 143: feat: add native currency support to Panoptic V1.1

This PR implements native currency support with the following changes:

1. Renamed instances of `token0`, `token1` to `currency0`, `currency1` and `token` to `asset`.
2. Removed the `payable` modifier from `multicall` to prevent bugs related to reuse of `msg.value`.
3. Made `deposit`, `mint`, and `deployNewPool` functions `payable`, implementing proper settlement flow for native currencies with refund handling.
4. Made `liquidate` `payable`, implementing settlement logic for native token liquidations with proper bonus handling.

Due to ordering, native currency is only passed on for `currency0`. Any native currency is refunded to the liquidator when settling liquidations if there are no negative bonuses.

A potential reentrancy vulnerability during liquidation settlement was identified and reported in finding "Potential reentrancy vulnerability in `settleLiquidation`".

4.1.2 PR 144: fix: increase capital requirement for liquidity addition DoS

This PR removes the problematic full-range liquidity add in the `PanopticFactory` and implements further restrictions on the enforced tick range in the `SemiFungiblePositionManager` that yield a capital requirement approximately equal to the greater of a default token value set or a percentage of the token supply. This PR was created to prevent a liquidity addition DoS.

This PR includes the implementation of the following functions:

- **Logarithm Calculations:** The PR implements approximate logarithm calculations through the `log_Sqrt1p0001MantissaRect` function:

```
/// @notice Approximates the absolute value of log base `sqrt(1.0001)` for a number in (0, 1) (`argX128/2^128`) with
↪  `precision` bits of precision.
/// @param argX128 The Q128.128 fixed-point number in the range (0, 1) to calculate the log of
/// @param precision The bits of precision with which to compute the result, max 63 (`err <|approx 2^-precision *
↪  log2(sqrt(1.0001))^-1`)
/// @return The absolute value of log with base `sqrt(1.0001)` for `argX128/2^128`
function log_Sqrt1p0001MantissaRect(
    uint256 argX128,
    uint256 precision
) internal pure returns (uint256) {
    unchecked {
        // =[log2(x)] =MSB(x)
        uint256 log2_res = FixedPointMathLib.log2(argX128);

        // Normalize argX128 to [1, 2)
        // x_normal = x / 2^[log2(x)]
        // = 1.a_1a_2a_3... = 2^(0.b_1b_2b_3...)
        // log2(x_normal) = log2(x / 2^\left\lfloor log2(x)\right\rfloor)
```

```
        // log2(x_normal) = log2(x) - log2(2^\left\lfloor log2(x)\right\rfloor)
        // log2(x_normal) = log2(x) - \left\lfloor log2(x)\right\rfloor
        // log2(x) = log2(x_normal) + \left\lfloor log2(x)\right\rfloor
        argX128 <<= (127 - log2_res);

        // =[log2(x)] * 2^64
        log2_res = (128 - log2_res) << 64;

        // log2(x_normal) = 0.b_1b_2b_3...
        // x_normal = (1.a_1a_2a_3...) = 2^(0.b_1b_2b_3...)
        // x_normal^2 = (1.a_1a_2a_3...)^2 = (2^(0.b_1b_2b_3...))^2
        // = 2^(0.b_1b_2b_3... * 2)
        // = 2^(b_1 + 0.b_2b_3...)
        // if b = 1, renormalize x_normal^2 to [1, 2):
        // 2^(b_1 + 0.b_2b_3...) / 2^b_1 = 2^((b_1 - 1).b_2b_3...)
        // = 2^(0.b_2b_3...)
        // error = [0, 2^-n)
        uint256 iterBound = 63 - precision;
        for (uint256 i = 63; i > iterBound; i--) {
            argX128 = (argX128 ** 2) >> 127;
            uint256 bit = argX128 >> 128;
            log2_res -= bit << i;
            argX128 >>= bit;
        }

        // log\sqrt _1._1(x) = log2(x) / log2(\sqrt 1.0001)
        // 2^64 / log2(\sqrt 1.0001)  255738959000112593413423
        return (log2_res * 255738959000112593413423) / 2 ** 128;
    }
}
```

The function computes the absolute value of the logarithm base \sqrt 1.0001 of a given number `argX128` in the range `(0, 1)` using Q128.128 fixed-point format:

1. Finds MSB position using `FixedPointMathLib.log2`.
2. Normalizes input to $[1, 2)$ range through bit shifting.
3. Iteratively calculates fractional bits up to specified precision.
4. Converts result to the correct base using the scaling factor 255738959000112593413423.

- **Details:** Q128.128 Fixed-Point Format: In this format, a 256-bit unsigned integer represents a real number where:
  - The most significant 128 bits represent the integer part (which will be zero in this case since our number is in `(0,1)`).
  - The least significant 128 bits represent the fractional part.

    ```
    uint256 log2_res = FixedPointMathLib.log2(argX128);
    ```

    Finds the position of the most significant bit (MSB) of `argX128`.

    ```
    argX128 <<= (127 - log2_res);
    ```

    By shifting left `(127 - log2_res)` bits, we move the MSB to position 127. This normalization step adjusts `argX128` to represent a number in the range `[1, 2)` in fixed-point format.

    ```
    // =[log2(x)] * 2^64
    log2_res = (128 - log2_res) << 64;
    ```

    Used to calculate the integer part of `|log2(x)|` scaled by `2^64` for Q64.64 fixed-point format. `(128 - log2_res)` gives the absolute value of the integer part of `log2(x)`. Shifting left by 64 bits scales this value to fixed-point format with 64 bits for the fractional part.

    ```
    uint256 iterBound = 63 - precision;
    for (uint256 i = 63; i > iterBound; i--) {
        argX128 = (argX128 ** 2) >> 127;
        uint256 bit = argX128 >> 128;
        log2_res -= bit << i;
        argX128 >>= bit;
    }
    ```

    Iteratively calculate each bit of the fractional part of `|log2(x)|` up to the specified precision.

    * `i`: Represents the current bit position in the fractional part (from most significant to least).
    * `iterBound`: Determines when to stop the loop based on the desired precision.

```
    argX128 = (argX128 ** 2) >> 127;
```

Square `argX128` and adjust it back to fixed-point format.

```
    uint256 bit = argX128 >> 128;
```

Extract the new MSB (the integer part after squaring).

```
    log2_res -= bit << i;
```

If the MSB is 1, subtract `1 << i` from `log2_res` to set the corresponding fractional bit.

```
    argX128 >>= bit;
```

If the MSB was 1, divide `argX128` by 2 to keep it in the range `[1, 2)`. This method is similar to computing logarithms using the identity: `log2(x) = k + log2(y)`, where `x = y * 2^k` and `y ∈ [1, 2)`. The fractional part is determined by iterative squaring and checking if the result exceeds 2.

```
    // log\sqrt 1.1(x) = log2(x) / log2(\sqrt 1.0001)
    // 2^64 / log2(\sqrt 1.0001) = 255738959000112593413423
    return (log2_X64 * 255738959000112593413423) / 2 ** 128;
```

The iterative process in the loop has computed `log2(x_normal)`, but it's stored in `log2_X64` in Q64.64 format. To convert a logarithm from one base to another, the change of base formula is used:

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

* `x` is the input value we took the log of.
* `b` is the base we initially used, which is 2 (since we calculated `log2(x)`).
* `a` is the new base we want, which is \sqrt 1.0001.

`255738959000112593413423` is an approximation of:

$$\frac{2^{64}}{\log 2\left(\sqrt{1.0001}\right)}$$

By multiplying by the scaling constant `255738959000112593413423` to convert from base 2 to base \sqrt 1.0001 we scale the result by an additional `2^64`, pushing it to Q128.128 format. Basically, the operation below is being performed but simply converting `log2_X64` first into Q128.128 format.

$$\log_{\sqrt{1.0001}}(x) = \frac{\log 2(x)}{\log 2(\sqrt{1.0001})}$$

The final division by `2^128` translates the Q128.128 representation to a plain decimal format.

- **Maximum Liquidity Per Tick Calculations:**
  - **UniswapV3 (v1.0) Implementation**

```
    /// @notice Computes the maximum liquidity that is allowed to reference any given tick in a Uniswap V3 pool
↪   with `tickSpacing`.
    /// @param tickSpacing The spacing between initializable ticks in the Uniswap V3 pool
    /// @return The maximum liquidity that can reference any given tick in the Uniswap V3 pool
    function getMaxLiquidityPerTick(int24 tickSpacing) internal pure returns (uint128) {
        unchecked {
            return type(uint128).max / uint24((Constants.MAX_V3POOL_TICK / tickSpacing) * 2 + 1);
        }
    }
```

Divides the maximum possible liquidity value by the total number of ticks, ensuring that the cumulative liquidity across all ticks does not exceed the maximum `uint128` value. Resolves to the following formula:

$$\text{maxLiquidityPerTick} = \frac{2^{128} - 1}{\left(\frac{2 \times \text{MAX\_TICK}}{\text{tickSpacing}}\right) + 1}$$

* UniswapV3 implementation:

```
    /// @notice Derives max liquidity per tick from given tick spacing
    /// @dev Executed within the pool constructor
    /// @param tickSpacing The amount of required tick separation, realized in multiples of `tickSpacing`
    ///     e.g., a tickSpacing of 3 requires ticks to be initialized every 3rd tick i.e., ..., -6, -3, 0,
↪  3, 6, ...
    /// @return The max liquidity per tick
    function tickSpacingToMaxLiquidityPerTick(int24 tickSpacing) internal pure returns (uint128) {
        int24 minTick = (TickMath.MIN_TICK / tickSpacing) * tickSpacing;
        int24 maxTick = (TickMath.MAX_TICK / tickSpacing) * tickSpacing;
        uint24 numTicks = uint24((maxTick - minTick) / tickSpacing) + 1;
        return type(uint128).max / numTicks;
    }
```

In addition to manual analysis, the functions were differentially tested to be equivalent.

– **UniswapV4 (v1.1) Implementation**

```
/// @notice Computes the maximum liquidity that is allowed to reference any given tick in a Uniswap V4 pool
↪  with `tickSpacing`.
/// @param tickSpacing The spacing between initializable ticks in the Uniswap V4 pool
/// @return maxLiquidityPerTick The maximum liquidity that can reference any given tick in the Uniswap V4 pool
function getMaxLiquidityPerTick(
    int24 tickSpacing
) internal pure returns (uint128 maxLiquidityPerTick) {
    int24 MAX_TICK = Constants.MAX_V4POOL_TICK;
    assembly {
        // Equivalent to type(uint128).max/(floor(MAX_TICK/tickSpacing) - floor(MIN_TICK/tickSpacing) + 1)
        maxLiquidityPerTick := div(
            MAX_UINT128,
            add(add(mul(div(MAX_TICK, tickSpacing), 2), gt(mod(MAX_TICK, tickSpacing), 0)), 1)
        )
    }
}
```

Which resolves to this formula:

$$\text{maxLiquidityPerTick} = \frac{2^{128} - 1}{2 \cdot \left\lfloor \frac{\text{MAX\_TICK}}{\text{tickSpacing}} \right\rfloor + (\text{MAX\_TICK} \bmod \text{tickSpacing} > 0) + 1}$$

The function divides `MAX_TICK` by `tickSpacing`, determining how many whole `tickSpacing` intervals fit within `MAX_TICK`. Then it multiplies the result by 2 to account for both positive and negative ticks (covering the range from minimum to maximum tick). It then checks if there's a remainder when dividing `MAX_TICK` by `tickSpacing`, adding 1 if so. Finally, the function divides the maximum possible `uint128` value (`MAX_UINT128`) by this adjusted tick range. This division yields the maximum liquidity that can be associated with each tick, ensuring it's spread out in accordance with the `tickSpacing` constraints.

* UniswapV4 implementation:

```
    /// @notice Derives max liquidity per tick from given tick spacing
    /// @dev Executed when adding liquidity
    /// @param tickSpacing The amount of required tick separation, realized in multiples of `tickSpacing`
    ///     e.g., a tickSpacing of 3 requires ticks to be initialized every 3rd tick i.e., ..., -6, -3, 0,
↪  3, 6, ...
    /// @return result The max liquidity per tick
    function tickSpacingToMaxLiquidityPerTick(int24 tickSpacing) internal pure returns (uint128 result) {
        // Equivalent to:
        // int24 minTick = (TickMath.MIN_TICK / tickSpacing);
        // if (TickMath.MIN_TICK  % tickSpacing != 0) minTick--;
        // int24 maxTick = (TickMath.MAX_TICK / tickSpacing);
        // uint24 numTicks = maxTick - minTick + 1;
        // return type(uint128).max / numTicks;
        int24 MAX_TICK = TickMath.MAX_TICK;
        int24 MIN_TICK = TickMath.MIN_TICK;
        // tick spacing will never be 0 since TickMath.MIN_TICK_SPACING is 1
        assembly ("memory-safe") {
            tickSpacing := signextend(2, tickSpacing)
            let minTick := sub(sdiv(MIN_TICK, tickSpacing), slt(smod(MIN_TICK, tickSpacing), 0))
            let maxTick := sdiv(MAX_TICK, tickSpacing)
            let numTicks := add(sub(maxTick, minTick), 1)
            result := div(sub(shl(128, 1), 1), numTicks)
        }
    }
```

In addition to manual analysis, the functions were differentially tested to be equivalent.

- **Min/Max Enforced Tick Calculation**

  The implementation calculates minimum and maximum tick bounds which require a given amount of capital to entirely fill the next tick.

```
function getApproxTickWithMaxAmount(
    uint256 amount,
    int24 tickSpacing,
    uint256 maxLiquidityPerTick
) internal pure returns (int24) {
    unchecked {
        // abs(max_error)  2^-13 * log2(|sqrt 1.0001)  -1.70234
        return
            int24(
                int256(
                    Math.log_Sqrt1p0001MantissaRect(
                        Math.mulDivCapped(
                            amount,
                            2 ** 224,
                            (maxLiquidityPerTick *
                                (Math.getSqrtRatioAtTick(tickSpacing) - 2 ** 96)),
                            128
                        ),
                        13
                    )
                )
            );
    }
}
```

  The implementation approximates the formula:

$$i = \log_{\sqrt{1.0001}} \left(\frac{C_{1\ fill}}{L_{maxTick}(1.0001^{\frac{s}{2}} - 1)} \right)$$

  - **Derivation for Max Tick**:

    Previous formulas:

$$L_{maxTick} = \frac{2^{128} - 1}{n_{ticks}} = \frac{2^{128} - 1}{887272/s_{tick} \cdot 2 + 1}$$

$$\sqrt{p(i)} = \sqrt{1.0001^i}$$

    Variables:

    * `amount`: $C_{1\ \text{fill}}$.
    * `tickSpacing`: s.
    * `maxLiquidityPerTick`: L_{maxTick}.
    * `tick`: i.

$$\begin{aligned} C_{1\ \text{fill}} &= L_{\text{maxTick}} \cdot \Delta\sqrt{p} \\ &= L_{\text{maxTick}} \cdot (\sqrt{p(i+s)} - \sqrt{p(i)}) \\ &= L_{\text{maxTick}} \cdot (1.0001^{\frac{i+s}{2}} - 1.0001^{\frac{i}{2}}) \\ &= L_{\text{maxTick}} \cdot 1.0001^{\frac{i}{2}}(1.0001^{\frac{s}{2}} - 1) \end{aligned}$$

$$\frac{C_{1\ \text{fill}}}{L_{\text{maxTick}}(1.0001^{\frac{s}{2}} - 1)} = 1.0001^{\frac{i}{2}}$$

$$2\log_{1.0001}\left(\frac{C_{1\ \text{fill}}}{L_{\text{maxTick}}(1.0001^{\frac{s}{2}} - 1)} \right) = i$$

- **Derivation for Min Tick:**

  Variables:

  - `amount`: $C_{0\ \text{fill}}$.

- `tickSpacing`: s.
- `maxLiquidityPerTick`: $L_{maxTick}$.
- `tick`: i.

$$
\begin{aligned}
C_{0\ \text{fill}} &= L_{maxTick} \cdot \Delta \frac{1}{\sqrt{p}} \\
&= L_{maxTick} \cdot \left(\frac{1}{\sqrt{p(i-s)}} - \frac{1}{\sqrt{p(i)}}\right) \\
&= L_{maxTick} \cdot \left(1.0001^{-\frac{i-s}{2}} - 1.0001^{-\frac{i}{2}}\right) \\
&= L_{maxTick} \cdot 1.0001^{-\frac{i}{2}}\left(1.0001^{\frac{s}{2}} - 1\right) \\
\frac{C_{0\ \text{fill}}}{L_{maxTick}\left(1.0001^{\frac{s}{2}} - 1\right)} &= 1.0001^{-\frac{i}{2}} \\
-2\log_{1.0001}\left(\frac{C_{0\ \text{fill}}}{L_{maxTick}\left(1.0001^{\frac{s}{2}} - 1\right)}\right) &= i
\end{aligned}
$$

The arguments passed into the function implementation are the desired cost `amount`, the `tickSpacing` parameter (retrieved from a pool's key parameter), and the `maxLiquidityPerTick` parameter which was analyzed previously.

The `amount` parameter is computed by multiplying the currency's total supply with a factor while enforcing a minimum tick fill cost. This restriction enforces a capital requirement approximately equal to the greater of a default token value set or a percentage of the token supply.

```
  /// @notice The multiplier, in basis points, to apply to the token supply and set as the minimum enforced tick fill
↪   cost if greater than `MIN_ENFORCED_TICKFILL_COST`.
  uint256 internal immutable SUPPLY_MULTIPLIER_TICKFILL;

  /// @notice The approximate minimum amount of tokens it should require to fill `maxLiquidityPerTick` at the minimum
↪   and maximum enforced ticks.
  uint256 internal immutable MIN_ENFORCED_TICKFILL_COST;

  uint256 amount = Math.max(
      MIN_ENFORCED_TICKFILL_COST,
      (IERC20Partial(Currency.unwrap(key.currency1)).totalSupply() * SUPPLY_MULTIPLIER_TICKFILL) / 10_000
  )
```

4.1.3 PR 148: feat: improve premium settlement, force exercise, and haircut APIs

The main purpose is to improve subgraph sync issues by providing more detailed tracking for premium settlement states, in particular for handling premium haircuts correctly.

The PR adds the following event changes to allow tracking of per-chunk settlement states:

- Emit premium by leg in `OptionBurned`.
- Emit leg in `PremiumSettled` event for `settleLongPremium(user, tokenId, leg)`.
- Emit a `PremiumSettled` event with negative for every leg haircut in `haircutPremia` during a liquidation.
- `forceExercise` is no longer working with the exercised position in the format `TokenId[touchedId]` and now it operates on `touchedId` directly.

`contracts/CollateralTracker.sol`

1. **Function Return Values:**
   - `takeCommissionAddData` and `_getExchangedAmount` now both return an additional variable: `commission`.

   The `commission` calculated in `_getExchangedAmount` is now first computed as the sum of the base rate commission and the ITM spread. The underlying calculation and rounding for the first return parameter `exchangedAmount` remains the same up to the order of operations.

contracts/PanopticPool.sol

1. **Event Parameters**:
    - Added `legIndex` to the `PremiumSettled` event.
    - Updated `LeftRightSigned premia` to `LeftRightSigned[4] premiaByLeg` in the `OptionBurnt` event.
    - Added `balanceData` and `commissions` parameters in the `OptionMinted` event.
    - `positionSize` and `poolUtilizations` parameters are now packed inside the `balanceDelta` along with the `tickData`.

    The `_mintOptions`, `_burnOptions` and `settleLongPremium` function's logic which emit the `OptionMinted`, `OptionBurnt` and `PremiumSettled` event has not changed other than the event emission.
2. **Function Return Values:**
    - `_mintInSFPMAndUpdateCollateral` and `_payCommissionAndWriteData` now return `commissions` as an additional parameter.
    - `_burnAndHandleExercise` now does not return `realizedPremia` anymore.

    The `_mintInSFPMAndUpdateCollateral`, `_payCommissionAndWriteData` and `_burnAndHandleExercise` function's logic has not changed aside from the added `commissions` and the removed `realizedPremia` return value.
3. **Premium Settlement Logic:**
    - Updated `premiaByLeg` in `_updateSettlementPostBurn` to reflect settled premiums for each leg.

    `premiaByLeg` only requires to be updated when the leg is short (`tokenId.isLong(leg) == 0`) as long positions always pay the full premium amount.
4. **Force Exercise Logic:**
    - Simplified `forceExercise` function's token ID parameter from array of length 1 to single variable.
    - Modified call to `_burnOptions` instead of `_burnAllOptionsFrom`.

    The call to `_burnOptions` for the single `itokenId` is equivalent to calling `_burnAllOptionsFrom` while passing in a single element array of token IDs.

contracts/SemiFungiblePositionManager.sol

1. **Variable Updates:**
    - Changed `amountToCollect`'s type from `LeftRightSigned` to `LeftRightUnsigned`.

    `PanopticMath.subRect` now returns a `LeftRightUnsigned` as the value is always positive.
2. **Logic Updates for Collecting and Writing Position Data:**
    - `LeftRightUnsigned amountToCollect` is now unsafely cast to `LeftRightSigned` before `movedInLeg` is subtracted.
    - The result is then unsafely cast back to `LeftRightUnsigned`.

    The initial cast should not result in any overflow as `amountToCollect` is retrieved from `subRect(LeftRightSigned, LeftRightSigned)`. The second cast is safe under the assumption that `movedInLeg` is strictly negative. The result of the subtraction will therefore be positive.

contracts/libraries/PanopticMath.sol

1. **New Import:**
    - Added `import {PanopticPool} from "@contracts/PanopticPool.sol";`.
2. **Variable Updates:**
    - Replaced separate `haircut0` and `haircut1` variables with a single `LeftRightSigned haircutBase`.

- Replaced separate `settled0` and `settled1` variables with a single `LeftRightSigned haircutAmounts`.
- Introduced `LeftRightUnsigned haircutTotal` for tracking cumulative haircut amounts.

3. **Haircut Premia Calculation Updates:**
   - Optimized the for loop in `haircutPremia` to continue if premias by leg is zero.
   - Updated haircut and settled token amounts calculations using the new `haircutBase`, `haircutAmounts` and `haircutTotal`.
   - `haircutAmounts` now uses the `LeftRightSigned`'s `sub` function handling overflow.

   The skip in `haircutPremia` can be done safely if `_premiasByLeg` is zero, as this results in a no-op when calculating the `haircutAmounts` and the `settledTokens`. Note that this also skips the event emission.

   The calculations surrounding `haircutBase` were found to be vulnerable by unsafe casting if the values could turn negative. Similarly, `premiasByLeg` must always be negative to be safely cast.

   Panoptic:

   > ... this value [`haircutBase`] should always be positive. You can verify this by analyzing the haircut/token swap branches where it is set
   >
   > ... long legs pay premium to sellers so their premium [`premiasByLeg`] will always be negative
   >
   > ... the full, unhaircut premium amount for each leg is already emitted in the burn event, so the event only needs to be emitted there if the original premium amount is reduced through a haircut

4. **Event Emission:**
   - Emitted `PanopticPool.PremiumSettled` event with new parameters to reflect updated haircut amounts.

5. **Exercise Update:**
   - Replaced early calls to `exercise` for non-zero `haircut0` and `haircut1` amounts to cumulative `haircutTotal` left and right values.

   The logic deciding when `exercise` is called for a liquidatee has changed slightly as it now depends on the cumulative haircut amounts.

   Panoptic:

   > Previously the tokens would be settled using the target haircut amount, which would then be prorated across all chunks rounding up and deducted. This means that the total amount of premium haircut from sellers can slightly exceed the amount returned to the liquidatee, which is a slight but benign leakage of value to rounding.
   >
   > However, now that we're emitting haircut amounts, this could cause discrepancies at the indexer level between settled tokens tracking for chunks and asset supply tracking (which is why we implemented this change to ensure both those deltas match).

`contracts/types/LeftRight.sol`

1. **Return Value Update:**
   - The `subRect` function now returns `LeftRightUnsigned` instead of `LeftRightSigned`.

   As both `leftSlot` and `rightSlot` are computed as `Math.max(x, 0)` these can be safely cast to a `uint128` value.

4.1.4 Commit dab4a5bb

The maximum cost for liquidating all a user's open positions was found to be too high. This commit restricts the open user position by limiting a user's total amount of open legs.

`contracts/CollateralTracker.sol`

1. **Position Check Updates:**
    - Replaced restrictions in `transfer`, `transferFrom` and `maxRedeem`, `maxWithdraw` involving user's open position counts to number of open legs.

    A user is unable to have open positions without any open legs. Therefore the new checks are equivalent.

`contracts/PanopticPool.sol`

1. **Variable Updates:**
    - Updated maximum open positions constant from 32 to 35 maximum open legs.

    A position can contain up to 4 legs. This reduces the maximum number of legs from 35 x 4 = 140 to 35.
2. **Open Leg Count Updates:**
    - Updated check in `_updatePositionsHash` to restrict number of open legs instead of open positions.

`contracts/libraries/PanopticMath.sol`

1. **Updates to Position Hash:**
    - `updatePositionsHash` now adds or subtracts the number of open legs of the token ID in the hash's upper byte.