Security Review Report
NM-0701 Panoptic V2

NETHERMIND
SECURITY

(January 12, 2026)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 4

4 System Overview 5
- 4.1 Core Architecture . . . 5
- 4.2 Liquidity Mechanics and Leverage . . . 5
- 4.3 Pricing Model: Streaming Premia . . . 6
- 4.4 Protocol Health and Maintenance . . . 6
- 4.5 Zero-Width Positions (Credit Lines) . . . 6
- 4.6 System Design Constraints . . . 6

5 Risk Rating Methodology 7

6 Issues 8
- 6.1 [High] Decimal precision mismatch in synthetic position collateral calculation leads to undercollateralization or DoS . . . 8
- 6.2 [Medium] Arithmetic overflow in `_updateSettlementPostBurn` causes incorrect `grossPremiumLast` calculation for token0 . . . 9
- 6.3 [Medium] Forced exercises might revert during market volatility due to incorrect casting in `getRefundAmounts(...)` . . . 10
- 6.4 [Medium] Stale collateral data in `_liquidate` causes protocol loss socialization by bypassing premium haircuts . . . 12
- 6.5 [Medium] The `_settleLongPremium(...)` function double counts the shortage when settling multiple long legs . . . 14
- 6.6 [Low] Attackers can force exercise 0-width long positions with minimal fees to increase victim's `netBorrows` . . . 16
- 6.7 [Low] The `exerciseCost(...)` function incorrectly considers in-range positions as out-of-range at the lower tick boundary . . . 17
- 6.8 [Info] The `SFPM`'s `registerTokenTransfer(...)` prevents transferring 0-width positions due to empty liquidity state . . . 18
- 6.9 [Best Practices] Rounding directions during bonus calculations in `getLiquidationBonus(...)` favor the liquidator . . . 19
- 6.10 [Best Practices] The `MAX_TICK_DELTA_SUBSTITUTION` constant is defined but not used . . . 20
- 6.11 [Best Practices] The `_burn(...)` function uses an unchecked block for `_internalSupply` decrement which could silently underflow . . . 21
- 6.12 [Best Practices] The `dispatchFrom(...)` function contains an unused local variable . . . 22
- 6.13 [Best Practices] The `validateIsExercisable(...)` function is unused and contains outdated logic . . . 23
- 6.14 [Best Practices] The overloaded `withdraw(...)` function in `CollateralTracker` bypasses the minimum asset retention mechanism . . . 24

7 Documentation Evaluation 25

8 Test Suite Evaluation 26
- 8.1 Tests Coverage . . . 26
- 8.2 Automated Tools . . . 26
  - 8.2.1 AuditAgent . . . 26

9 About Nethermind 27

1 Executive Summary

This document presents the security review conducted by Nethermind Security for the Panoptic Protocol's V2 core smart contracts. Panoptic is an oracle-less, permissionless options protocol built on top of the Uniswap V3 and V4 concentrated liquidity automated market makers (AMM). While the protocol architecture supports both versions, this review focuses strictly on the Uniswap V3 integration. It enables the creation of perpetual options by effectively lending and borrowing Uniswap V3 liquidity positions. By replacing traditional upfront premiums with a novel "Streamia" pricing model, Panoptic allows fees to accumulate block-by-block based on liquidity utilization, facilitating the creation of capital-efficient, multi-leg option strategies in a single transaction. The security review covered the core smart contracts, with a primary focus on the logic that was changed from the V1 and V1.1 system to the V2 version.

The audit comprises 7948 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase and (b) automated analysis tools. **Along this document, we report 14 points of attention** where one is classified as `High`, four are classified as `Medium`, two are classified as `Low`, and seven are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 presents the system overview. Section 4 discusses the risk rating methodology. Section 5 details the issues. Section 6 discusses the documentation provided by the client for this audit. Section 7 presents the test suite evaluation and automated tools used. Section 8 concludes the document.

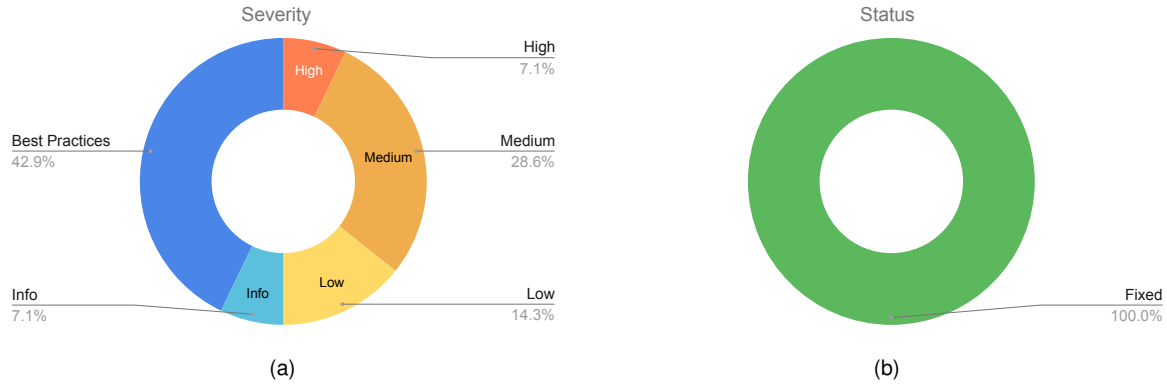

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (4), **Low** (2), **Undetermined** (0), **Informational** (1), **Best Practices** (6). **Distribution of status: Fixed** (14), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	December 18, 2025
Final Report	January 12, 2026
Initial Commit	132c751f01a0f7225a64c282fb5c5a327b0cb372
Final Commit	4024dbc71c83d3ca64ed786591ddb9e9362c7041
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/RiskEngine.sol	1177	556	47.2%	169	1902
2	contracts/CollateralTracker.sol	814	401	49.3%	173	1388
3	contracts/PanopticFactory.sol	182	85	46.7%	39	306
4	contracts/SemiFungiblePositionManager.sol	876	589	67.2%	155	1620
5	contracts/PanopticPool.sol	1403	532	37.9%	208	2143
6	contracts/libraries/SafeTransferLib.sol	78	34	43.6%	13	125
7	contracts/libraries/Math.sol	836	451	53.9%	156	1443
8	contracts/libraries/PanopticMath.sol	852	401	47.1%	111	1364
9	contracts/libraries/EfficientHash.sol	56	26	46.4%	8	90
10	contracts/libraries/FeesCalc.sol	52	107	205.8%	5	164
11	contracts/libraries/Errors.sol	45	48	106.7%	38	131
12	contracts/libraries/InteractionHelper.sol	59	33	55.9%	7	99
13	contracts/libraries/CallbackLib.sol	26	18	69.2%	4	48
14	contracts/libraries/Constants.sol	17	24	141.2%	13	54
15	contracts/base/Multicall.sol	23	15	65.2%	4	42
16	contracts/base/FactoryNFT.sol	335	59	17.6%	23	417
17	contracts/base/MetadataStore.sol	17	15	88.2%	3	35
18	contracts/types/LiquidityChunk.sol	93	83	89.2%	15	191
19	contracts/types/PoolData.sol	36	40	111.1%	9	85
20	contracts/types/TokenId.sol	354	209	59.0%	47	610
21	contracts/types/Pointer.sol	45	26	57.8%	13	84
22	contracts/types/LeftRight.sol	216	98	45.4%	46	360
23	contracts/types/PositionBalance.sol	142	85	59.9%	19	246
24	contracts/tokens/ERC1155Minimal.sol	136	83	61.0%	37	256
25	contracts/tokens/ERC20Minimal.sol	71	61	85.9%	30	162
26	contracts/tokens/interfaces/IERC20Partial.sol	7	20	285.7%	4	31
	Total	**7948**	**4099**	**51.6%**	**1349**	**13396**

*The audit prioritized the analysis of changes introduced in the v2 upgrade, specifically targeting new features, architectural shifts, and their integration with the existing system. The scope encompassed the broader codebase, including established files, to ensure auditors maintained a holistic view of the protocol's security posture and system-wide invariants. Note, however, that the cryptographic primitives underlying the new position hashing mechanism, including but not limited to the homomorphic hash implementation, were explicitly excluded from the primary focus of this engagement.

3 Summary of Issues

	Finding	Severity	Update
1	Decimal precision mismatch in synthetic position collateral calculation leads to undercollateralization or DoS	High	Fixed
2	Arithmetic overflow in `_updateSettlementPostBurn` causes incorrect `grossPremiumLast` calculation for token0	Medium	Fixed
3	Forced exercises might revert during market volatility due to incorrect casting in `getRefundAmounts(...)`	Medium	Fixed
4	Stale collateral data in `_liquidate` causes protocol loss socialization by bypassing premium haircuts	Medium	Fixed
5	The `_settleLongPremium(...)` function double counts the shortage when settling multiple long legs	Medium	Fixed
6	Attackers can force exercise 0-width long positions with minimal fees to increase victim's `netBorrows`	Low	Fixed
7	The `exerciseCost(...)` function incorrectly considers in-range positions as out-of-range at the lower tick boundary	Low	Fixed
8	The `SFPM` 's `registerTokenTransfer(...)` prevents transferring 0-width positions due to empty liquidity state	Info	Fixed
9	Rounding directions during bonus calculations in `getLiquidationBonus(...)` favor the liquidator	Best Practices	Fixed
10	The `MAX_TICK_DELTA_SUBSTITUTION` constant is defined but not used	Best Practices	Fixed
11	The `_burn(...)` function uses an unchecked block for `_internalSupply` decrement which could silently underflow	Best Practices	Fixed
12	The `dispatchFrom(...)` function contains an unused local variable	Best Practices	Fixed
13	The `validateIsExercisable(...)` function is unused and contains outdated logic	Best Practices	Fixed
14	The overloaded `withdraw(...)` function in `CollateralTracker` bypasses the minimum asset retention mechanism	Best Practices	Fixed

4 System Overview

The Panoptic V2 protocol is an oracle-less, permissionless options protocol built on top of the Uniswap V3 concentrated liquidity automated market maker (AMM). It enables the creation of perpetual options by moving liquidity between the Panoptic ecosystem and underlying Uniswap pools. Unlike traditional options which charge an upfront premium, Panoptic utilizes a novel "Streamia" pricing model where fees accumulate block-by-block based on the utilization of liquidity within specific price ranges. The system is designed to be composable, allowing users to mint complex option strategies with up to four legs in a single transaction.

4.1 Core Architecture

The protocol architecture is modular, separating position management, collateral tracking, and risk assessment into distinct contracts orchestrated by a central hub.

- **The Panoptic Pool**: This contract serves as the primary entry point for all user interactions, including minting and burning options, liquidations, and forced exercises. It maintains a "Positions Hash" for each user to gas-efficiently track active portfolios without storing the full state of every position on-chain.
- **Semi-Fungible Position Manager (SFPM)**: The SFPM is the engine that wraps Uniswap V3 positions. It functions as an ERC1155 ledger that tracks the ownership of Option Positions (TokenIds) rather than individual liquidity chunks. A single TokenId wraps up to four distinct legs. The SFPM internally decomposes this ID into operations on specific liquidity chunks (via `_createLegInAMM`) to manage the interaction with Uniswap. It distinguishes between `Net Liquidity` (funds actively deployed in the AMM) and `Removed Liquidity` (funds borrowed from the AMM to create long option positions) to calculate the streaming premium.
- **Collateral Tracking System**: Each Panoptic Pool is paired with two `CollateralTracker` contracts (one for token0 and one for token1) which function as ERC4626 vaults. These trackers manage user deposits, issue virtual shares, and handle the settlement of funds (`settleMint` and `settleBurn`) when users open or close positions.
- **Risk Engine**: This contract encapsulates the protocol's risk management logic. It contains an internal median oracle that stores an 8-slot observation queue to provide a manipulation-resistant price feed. The Risk Engine facilitates undercollateralized leverage by allowing users to control large liquidity positions while depositing only a fraction of the value as collateral. It dynamically calculates these requirements based on pool utilization and the "moneyness" of a position, performing solvency checks (`isAccountSolvent`) before allowing any account modification.

4.2 Liquidity Mechanics and Leverage

Conceptually, Panoptic functions as a lending market for Uniswap V3 liquidity positions. The protocol enables leverage because users do not need to fully capitalize the positions they create; instead, they direct the protocol's pooled funds while maintaining a fractional margin requirement.

- **Short Options (Leveraged Liquidity Provision)**: When a user mints a short option, they are not borrowing tokens into their wallet. Instead, they direct the Panoptic Pool to deploy its assets into a Uniswap V3 pool at a specific tick range on their behalf.
  - **Action**: The protocol *adds* liquidity to the AMM using pooled funds.
  - **Leverage**: The user acts as a leveraged Liquidity Provider (LP). The Risk Engine requires collateral significantly lower than the notional value of the position (e.g., varying based on moneyness and utilization), allowing for capital-efficient market making.
  - **Result**: The user earns fees (Streamia) from trading activity but is exposed to divergent loss (impermanent loss) if the price moves.
- **Long Options (Short-Selling Liquidity)**: When a user mints a long option, they effectively "borrow" the liquidity that a short seller has provided. Mechanically, this removes the liquidity from the Uniswap V3 pool and holds it within the Panoptic protocol.
  - **Action**: The protocol *removes* liquidity from the AMM.
  - **Leverage**: The user creates a liability by effectively short-selling the LP token. The Risk Engine requires far less collateral to hold this "short LP" position than the cost of buying the liquidity outright, providing significant capital efficiency for hedging or speculation.
  - **Result**: The user pays fees (Streamia) to the short sellers for the right to displace their liquidity. By removing the liquidity, the user ensures that the short seller cannot earn AMM swap fees, so the user must compensate them directly.

This structure creates an inverse relationship for closing positions: "Burning" (closing) a short position involves *removing* funds from Uniswap to repay the Panoptic Pool, whereas "Burning" a long position involves *adding* liquidity back into Uniswap to restore the pool's state.

4.3 Pricing Model: Streaming Premia

Panoptic replaces the Black-Scholes pricing model with **Streamia**, a path-dependent accumulation of fees. It means that options don't have an upfront cost; instead, fees build up gradually over time based on the asset's actual price movements.

- **Streamia Concept**: Premiums are calculated based on the spread between `Net Liquidity` (N, liquidity remaining in the AMM) and `Removed Liquidity` (R, liquidity taken by long buyers). The protocol tracks the swap fees collected by N to determine the "fair value" that R would have earned.
- **Accumulators**: The system maintains `s_accountPremiumOwed` (for long legs) and `s_accountPremiumGross` (for short legs). Short sellers earn a pro-rata share of the gross premium based on the total liquidity deposited, while long buyers pay based on the liquidity they removed.

4.4 Protocol Health and Maintenance

The system employs rigorous checks and maintenance functions to ensure solvency and proper fee distribution.

- **Liquidations**: If an account's collateral falls below the required maintenance margin, it becomes eligible for liquidation. The `liquidate` function burns the distressed user's positions and seizes their collateral to pay a bonus to the liquidator. To handle insolvency (protocol loss), the system employs a tiered defense. First, it attempts a `haircutPremia` mechanism, reducing the `s_settledTokens` accumulator for the specific liquidity chunk to claw back uncollected premiums. If this haircut is insufficient to cover the bad debt, the protocol mints new, unbacked shares in the `CollateralTracker` to the liquidator. This final backstop ensures the liquidator is compensated but effectively socializes the remaining loss across all Collateral Tracker liquidity providers, as the total supply of shares increases without a corresponding increase in underlying assets.
- **Forced Exercises**: To ensure liquidity can be retrieved from long positions (e.g., to allow a short seller to exit), the protocol allows for `forceExercise`. This enables a third party to close a user's long position, typically when it is out-of-range or when necessary to restore pool utilization. The cost is dynamic: expensive if the position is in-range (to protect the user) but cheap (e.g., 1 bps) if out-of-range.
- **Settling Long Premium**: The `settleLongPremium` function is a maintenance tool used to collect owed premiums from long positions without closing them. Since option sellers only realize their gains when they burn (close) their positions, this function allows the system (or users) to force long buyers to pay their accrued debts into the `s_settledTokens` accumulator. This ensures that when short sellers eventually exit, there are sufficient settled assets available to pay out their earned yield.

4.5 Zero-Width Positions (Credit Lines)

A distinct feature of Panoptic V2 is the support for **Zero-Width Positions**. These are defined with a width of 0, meaning they do not represent a range of liquidity in the AMM but rather a direct loan or credit.

- **Functionality**: A zero-width position treats the underlying asset purely as a borrowed sum (synthetic short) or a lent sum (synthetic long) without interacting with Uniswap ticks.
- **Collateral Treatment**:
  - **Short Zero-Width**: Represents *borrowing* assets from the pool. This creates a direct liability and requires high collateralization (>100% plus margin buffer).
  - **Long Zero-Width**: Represents *lending* assets to the pool. This acts as a credit position, effectively reducing the user's overall collateral requirements. The zero-width positions don't earn the regular interest. They can be used to reduce the interest paid on other positions.

4.6 System Design Constraints

Certain design choices impose specific constraints on the protocol's operation:

- **Liquidity Spread**: The protocol enforces a maximum utilization ratio ($R/N \leq 90\%$) via the `checkLiquiditySpread` logic. In low-liquidity pools, if a single "whale" short seller controls the majority of the liquidity, liquidating them might be blocked if it causes the R/N ratio to exceed the limit. In such cases, liquidators must first force-exercise long buyers to lower R (removed liquidity) before the short position can be successfully liquidated. This dependency creates a more complex, multi-step process for liquidators; delays in performing the prerequisite force exercises can extend the time a distressed account remains solvent, potentially allowing bad debt to accumulate.
- **Asset Compatibility**: The system's internal accounting assumes standard ERC20 behavior. Tokens with fee-on-transfer mechanisms or rebasing tokens are not supported and will result in accounting discrepancies within the `CollateralTracker` and `SFPM`. This design limitation mirrors the architecture of the underlying Uniswap V3 protocol, which similarly does not natively support non-standard token implementations.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Decimal precision mismatch in synthetic position collateral calculation leads to undercollateralization or DoS

File(s): contracts/RiskEngine.sol

Description: The `RiskEngine` contract calculates collateral requirements for various option strategies. Specifically, the `_getRequiredCollateralSingleLegPartner(...)` function handles multi-leg positions, such as synthetic stocks, where one leg is long and the other is short (`_isLong != isLongP`).

To determine the collateral requirement for such positions, the function calculates the requirement for both legs independently and takes the maximum value. However, it performs a raw integer comparison using `Math.max(...)` without normalizing the values for the differing market prices of the underlying tokens. This logic fails to accurately assess the economic value of the collateral requirement. The result is then assigned as the requirement for the token associated with the lower-indexed leg.

```
function _getRequiredCollateralSingleLegPartner(...) internal view returns (uint256) {
    // ...
    } else if (_isLong != isLongP) {
        // SYNTHETIC STOCK: different token types, one is long and the other is short
        return
        // return the largest collateral requirement of the two legs
        index < partnerIndex
            ? Math.max(
                _getRequiredCollateralSingleLegNoPartner(
                    tokenId,
                    index,
                    positionSize,
                    atTick,
                    poolUtilization
                ),
                // @audit-issue The comparison is done on raw values without normalization.
                // If tokens have different decimals (e.g. 18 vs 8), the comparison is flawed.
                // This is also incorrect for same decimal, but different value pairs e.g. DAI/WETH.
                _getRequiredCollateralSingleLegNoPartner(
                    tokenId,
                    partnerIndex,
                    positionSize,
                    atTick,
                    poolUtilization
                )
            )
            : 0;
    }
    // ...
}

```

While this issue exists for any two assets with different market values, the most apparent/visible case is when two assets have significantly different decimal precisions (e.g., WETH with 18 decimals and WBTC with 8 decimals) as as the raw integer scales will differ by orders of magnitude regardless of price.

This mismatch leads to two scenarios:

a. **Undercollateralization**: If the lower-indexed leg (`Leg 0`) is the high-precision token (e.g., WETH) and the higher-indexed leg (`Leg 1`) is the low-precision token (e.g., WBTC):
   - Assume `Leg 0` (WETH) requires `0.1 WETH` (`1e17` raw units);
   - Assume `Leg 1` (WBTC) requires `1 WBTC` (`1e8` raw units);
   - The function compares `1e17` vs `1e8`. Since `1e17 > 1e8`, it returns `1e17`;
   - The system requires `0.1 WETH` ($\approx \$200$) to secure a position involving `1 WBTC` ($\approx \$60,000$), leaving the protocol undercollateralized.

b. **Denial of Service (DoS)**: If the leg indexing is reversed, such that `Leg 0` is the low-precision token (WBTC) and `Leg 1` is WETH:
   - Assume `Leg 0` (WBTC) requires `0.1 WBTC` (`1e7` raw units);
   - Assume `Leg 1` (WETH) requires `10 WETH` (`1e19` raw units);
   - The function compares `1e7` vs `1e19`. It returns `1e19`;
   - The system sets the requirement to `1e19` units of the `Leg 0` token (WBTC). This equates to `100,000,000,000` WBTC, which likely exceeds the available balance, causing the transaction to revert.

The severity is assessed as High because the vulnerability combines Medium Likelihood (common pairs like WBTC/WETH or USDC/ETH inherently have decimal mismatches; however issue occurs only with specific position type - synthetic stocks) with High Impact. Specifically, the undercollateralization scenario allows users to obtain massive leverage with minimal capital, directly risking protocol insolvency, while the DoS scenario breaks certain functionality for specific pool configurations.

Recommendation(s): Consider refactoring the logic to correctly identify and choose the highest collateral requirement between the two legs, ensuring that the selection reflects the true economic risk rather than raw integer magnitude.

Status: Fixed.

Update from the client: Fixed in `5c49a61`.

6.2 [Medium] Arithmetic overflow in `_updateSettlementPostBurn` causes incorrect `grossPremiumLast` calculation for token0

File(s): `contracts/PanopticPool.sol`

Description: The `_updateSettlementPostBurn` function is responsible for updating the `s_grossPremiumLast` accumulator when a position is burned. This value is used to correctly track the premium entitlement of the remaining liquidity in the chunk.

The calculation involves adding the `legPremia` (amount settled for the burning position) to the historical accumulator, scaled by `2**64`. The implementations for token0 (right slot) and token1 (left slot) differ in the order of operations regarding casting.

For token1 (left slot), the `int128` value is cast to `int256` *before* multiplication:

```
)) + int256(legPremia.leftSlot()) * 2 ** 64,
```

However, for token0 (right slot), the multiplication occurs before the cast:

```
)) + int256(legPremia.rightSlot() * 2 ** 64),
```

`legPremia.rightSlot()` returns an `int128`. Since the literal `2**64` fits within an `int128`, Solidity performs the multiplication using `int128` arithmetic. The maximum value of `int128` is `2^127 - 1`. If `legPremia.rightSlot()` is greater than `2^63 - 1` (approximately `9.22e18`), the multiplication `legPremia.rightSlot() * 2**64` will exceed `2^127 - 1` and overflow.

Because this calculation is wrapped in an `unchecked` block, the overflow wraps to a negative number. Casting this negative `int128` to `int256` results in a large negative value, which is then subtracted from the accumulator sum. This often causes the result to be less than zero, which `Math.max` clamps to `0`.

Setting `s_grossPremiumLast` to 0 breaks the accounting for the remaining liquidity in that chunk. The claimable premium is calculated as `totalLiquidity * (grossAccumulator - grossPremiumLast)`.

Recommendation(s): Correct the order of operations for the token0 calculation to cast to `int256` before multiplying.

Status: Fixed.

Update from the client: The parenthesis has been moved to the correct location, see here.

6.3 [Medium] Forced exercises might revert during market volatility due to incorrect casting in `getRefundAmounts(...)`

File(s): contracts/RiskEngine.sol

Description: The `getRefundAmounts(...)` function in the `RiskEngine` contract is responsible for calculating token substitutions during forced exercises or premium settlements. It ensures that if the Exercisee (the `payor`) lacks sufficient tokens to cover the transaction costs or fees, the shortage is covered by the Exercisor (caller) in exchange for the other token.

In the context of Panoptic, the exercise fee represents the difference between the Oracle Price (Fair Value) and the Spot Price (Execution Price). This fee is typically negative because forced exercises occur against the user's will. The Liquidator compensates the User based on a base fee or a force exercise cost percentage. Even when exercising deep Out-of-the-Money (OTM) positions where the cost is low, the fee remains negative, meaning the Exercisor pays. A positive fee (where the User owes the Liquidator) only applies in specific scenarios where the Spot Price provides a better execution value than the Oracle Price.

However, the logic used to calculate the `balanceShortage` incorrectly handles positive fee values. Specifically, the code negates the signed fee value and immediately casts it to `uint128`.

```
function getRefundAmounts(
    address payor,
    LeftRightSigned fees,
    int24 atTick,
    CollateralTracker ct0,
    CollateralTracker ct1
) external view returns (LeftRightSigned) {
    uint160 sqrtPriceX96 = Math.getSqrtRatioAtTick(atTick);
    unchecked {
        // @audit-issue If fee is positive, it becomes negative and is then
        // casted to uint128. Due to two's complement, the result will become
        // 2**128 - fee, which is a large positive number. The balance
        // shortage will become a large negative number.
        int256 balanceShortage = int256(uint256(type(uint248).max))
            - int256(ct0.balanceOf(payor))
            - int256(ct0.convertToShares(uint128(-fees.rightSlot())));

        // @audit-issue The shortage adjustment logic won't be executed
        // since calculated shortage is negative.
        if (balanceShortage > 0) {
            return LeftRightSigned.wrap(0).addToRightSlot(
                int128(
                    fees.rightSlot()
                        - int256(
                            Math.mulDivRoundingUp(
                                uint256(balanceShortage),
                                ct0.totalAssets(),
                                ct0.totalSupply()
                            )
                        )
                )
            ).addToLeftSlot(
                int128(
                    int256(
                        PanopticMath.convert0to1RoundingUp(
                            ct0.convertToAssets(uint256(balanceShortage)),
                            sqrtPriceX96
                        )
                    ) + fees.leftSlot()
                )
            );
        }
        balanceShortage = int256(uint256(type(uint248).max))
            - int256(ct1.balanceOf(payor))
            - int256(ct1.convertToShares(uint128(-fees.leftSlot())));

        if (balanceShortage > 0) {
            // ...
        }
    }
    return fees;
}
```

If `fees.rightSlot()` is positive (e.g., `50`), `-fees.rightSlot()` becomes `-50`. Casting `-50` to `uint128` results in a very large positive number (approximately `2^128 - 50`). Consequently, the `balanceShortage` calculation subtracts this huge number, resulting in a large negative value.

Because `balanceShortage` is negative, the `if (balanceShortage > 0)` check fails, and the shortage handling logic is skipped. The function returns the full positive fee amount. Subsequently, the `CollateralTracker` will attempt to `transferFrom` the Exercisee's account to the Exercisor. If the Exercisee does not have sufficient balance, the transaction will revert.

This issue creates a Denial of Service (DoS) for forced exercises in specific volatility scenarios. These scenarios occur when the Spot Price and Oracle Price diverge significantly, such that one price is inside the position's tick range while the other is outside. This effectively blocks liquidators from force-exercising long positions. Force-exercising long positions is a necessary step to reduce "Removed Liquidity" and normalize liquidity ratios before liquidating large short positions. Consequently, this exposes the protocol to bad debt during market stress.

Recommendation(s): Consider implementing logic to correctly handle positive fees within the `getRefundAmounts(...)` function, ensuring that the casting logic supports both positive and negative fee values or skips the shortage check when the user is paying.

Status: Fixed.

Update from the client: This is the new code that correctly handles the fees' sign: here and here.

6.4 [Medium] Stale collateral data in `_liquidate` causes protocol loss socialization by bypassing premium haircuts

File(s): contracts/PanopticPool.sol

Description: The `_liquidate` function in `PanopticPool` is responsible for closing a distressed user's positions, seizing their collateral, and calculating protocol losses to apply necessary premium haircuts. The function first snapshots the user's margin details (collateral balance and requirements) using `getMargin`.

```
function _liquidate(...) internal returns (LeftRightSigned bonusAmounts) {
    // ...
    // @audit Snapshots user balance (tokenData.rightSlot) and requirements (tokenData.leftSlot)
    // tokenData.leftSlot includes margin requirement + interest owed
    // tokenData.rightSlot includes current asset balance (before interest payment)
    (tokenData0, tokenData1, ) = riskEngine().getMargin(...);
    // ...
```

After snapshotting, `_liquidate` calls `_burnAllOptionsFrom` to close the positions. This triggers `CollateralTracker.settleBurn`, which in turn calls `_accrueInterest`. If the user owes interest, `_accrueInterest` burns user shares to settle the debt, effectively reducing the user's asset balance.

```
function _burnOptions(...) internal ... {
    // ...
    // @audit Triggers settleBurn -> _accrueInterest
    // This burns shares from the user, reducing their actual collateral balance
    (LeftRightSigned longAmounts, LeftRightSigned shortAmounts) = PanopticMath.computeExercisedAmounts(...);
    collateralToken0().settleBurn(...);
    // ...

function _updateBalancesAndSettle(
    bool isCreation,
    address optionOwner,
    int128 longAmount,
    int128 shortAmount,
    int128 ammDeltaAmount,
    int128 realizedPremium
) internal returns (uint32, uint128, int128) {
    _accrueInterest(optionOwner, IS_NOT_DEPOSIT);
    //...
}
```

However, the subsequent call to `getLiquidationBonus` uses the stale `tokenData` snapshot taken before the interest payment.

```
// PanopticPool._liquidate(...)
// @audit Uses stale tokenData0/tokenData1 (pre-interest burn balances)
(bonusAmounts, collateralRemaining) = riskEngine().getLiquidationBonus(
    tokenData0,
    tokenData1,
    // ...
);
```

In `RiskEngine.getLiquidationBonus`, the `collateralRemaining` (used to determine protocol loss) is calculated using the stale balance: `balance - netPaid - bonus`. It does not account for the assets consumed by the interest payment during the burn.

```
// RiskEngine.getLiquidationBonus(...)
// ...
// @audit balance0 derives from stale tokenData0.rightSlot() (pre-interest burn)
int256 balance0 = int256(uint256(tokenData0.rightSlot())) -
    int256(uint256(shortPremium.rightSlot()));

// ...
// @audit paid0 includes the bonus and the net cost to close positions
int256 paid0 = bonus0 + int256(netPaid.rightSlot());

// ...
return (
    // ...
    // @audit collateralRemaining calculated as balance - paid
    // If actual balance < balance0 due to interest, this returns a false positive surplus
    LeftRightSigned.wrap(0).addToRightSlot(int128(balance0 - paid0)).addToLeftSlot(
        int128(balance1 - paid1)
    )
);
```

If the interest payment reduces the user's balance such that they cannot cover the liquidation costs (netPaid + bonus), the user is actually in a deficit (protocol loss). However, because `getLiquidationBonus` uses the higher pre-interest balance, it may incorrectly calculate a positive `collateralRemaining`.

This false surplus causes `PanopticMath.haircutPremia` to skip the haircut logic which is designed to claw back premiums from short sellers in the specific chunk to cover losses.

```
// PanopticMath.sol

function haircutPremia(...) external returns (LeftRightSigned) {
    // ...
    // @audit Protocol loss is detected if collateralRemaining is negative.
    // Because collateralRemaining was calculated with stale data, it might be positive here.
    // Thus, collateralDelta becomes 0, identifying no loss.
    int256 collateralDelta0 = -Math.min(collateralRemaining.rightSlot(), 0);
    int256 collateralDelta1 = -Math.min(collateralRemaining.leftSlot(), 0);

    // ...
    // @audit Logic to haircut short sellers is skipped because deltas are 0
    if (
        longPremium.rightSlot() < collateralDelta0 &&
        longPremium.leftSlot() > collateralDelta1
    ) {
        // ...
    }
}
```

This false surplus causes `PanopticMath.haircutPremia` to skip the haircut logic (which is designed to claw back premiums from short sellers in the specific chunk to cover losses).

```
// CollateralTracker.sol

function settleLiquidation(...) external onlyPanopticPool {
    // ...
    // @audit liquidator needs to be paid (bonus < 0 means protocol pays liquidator here)
    if (bonus < 0) {
        uint256 bonusAbs = uint256(-bonus);
        // ...
        uint256 liquidateeBalance = balanceOf[liquidatee];
        // @audit If user is insolvent (balance < bonus due to interest burn),
        // we wipe their balance but still mint the full bonus to the liquidator.
        if (type(uint248).max > liquidateeBalance) {
            balanceOf[liquidatee] = 0;
            unchecked {
                // @audit Creates unbacked shares, socializing the loss
                _internalSupply += type(uint248).max - liquidateeBalance;
            }
        }
        // ...
    }
}
```

Instead, the protocol realizes the loss when `settleLiquidation` is called; it wipes the user's balance and mints unbacked bonus shares to the liquidator, diluting all depositors in the `CollateralTracker` and socializing the loss that should have been isolated to the specific option sellers.

Recommendation(s): Consider updating the system to ensure the `tokenData` used for liquidation calculations reflects the post-interest-settlement state.

Status: Fixed.

Update from the client: See commit fixing that issue here.

6.5 [Medium] The `_settleLongPremium(...)` function double counts the shortage when settling multiple long legs

File(s): contracts/PanopticPool.sol, contracts/RiskEngine.sol

Description: The `_settleLongPremium(...)` function in the `PanopticPool` contract allows option sellers to settle unpaid premium for long legs. The process involves delegating virtual shares to the owner (inflating balance to ensure premia can be burned), iterating through legs to burn the owed premium via `settleBurn(...)`, and calculating if a refund is needed from the caller using `getRefundAmounts(...)`.

The issue arises from how `getRefundAmounts(...)` calculates the shortage. It determines the shortage by comparing the user's current balance against the inflation amount (`type(uint248).max`).

```
// contracts/RiskEngine.sol
function getRefundAmounts(
    address payor,
    LeftRightSigned fees,
    // ...
    CollateralTracker ct0,
    CollateralTracker ct1
) external view returns (LeftRightSigned) {
    // ...
    unchecked {
        // @audit-issue The shortage is calculated based on the
        // CURRENT balance of the payor.
        int256 balanceShortage = int256(uint256(type(uint248).max))
            - int256(ct0.balanceOf(payor))
            - int256(ct0.convertToShares(uint128(-fees.rightSlot())));

        // ... logic for token1 ...
    }
    // ...
}
```

Inside the loop in `_settleLongPremium(...)`, the user's balance is decreased in each iteration by the `settleBurn(...)` call. Because `getRefundAmounts` reads the live balance, the calculation of shortage for subsequent legs implicitly includes the premium burned for previous legs (since the balance is lower).

Since `refundAmounts` accumulates these values via `.add(...)`, the shortage created by earlier legs is added to the total multiple times.

For example, assuming a user has 0 shares initially and must settle two legs:

1. Leg 1 burns `10`. Shortage calculated is `10`. `refundAmounts` becomes `10`;
2. Leg 2 burns `20`. The user's balance is now `-30` relative to the inflation start point. `getRefundAmounts(...)` calculates the current shortage as `30` (Leg 1's `10` + Leg 2's `20`);
3. This `30` is added to `refundAmounts`. Total `refundAmounts` becomes `10 + 30 = 40`;
4. The actual total shortage required to be refunded should be `10 + 20 = 30`;

In general, the shortage inflation follows a cumulative pattern based on the order of leg iteration. If leg 1 owes premium `x` and leg 2 owes premium `y`, the first iteration calculates a shortage of `x`. The second iteration calculates a cumulative shortage of `x + y` because the balance has already decreased by `x`. This results in a total requested payment of `2x + y` instead of the correct `x + y`.

This results in the caller overpaying when settling positions with multiple long legs.

```
// contracts/PanopticPool.sol
function _settleLongPremium(...) internal {
    CollateralTracker ct0 = collateralToken0();
    // ...
    // @audit The protocol delegates some virtual shares to ensure the premia
    // can be burned, even if user does not have shares of this collateral token.
    // The shortage is provided by the caller.
    ct0.delegate(owner);
    ct1.delegate(owner);
    // ...
    LeftRightSigned refundAmounts;
    for (uint256 leg = 0; leg < tokenId.countLegs();) {
        if (tokenId.width(leg) != 0 && tokenId.isLong(leg) != 0) {
            // ... logic to calculate realizedPremia ...
            {
                // @audit User balance is decreased here via burn
                ct0.settleBurn(owner, 0, 0, 0, -realizedPremia.rightSlot());
                ct1.settleBurn(owner, 0, 0, 0, -realizedPremia.leftSlot());
                // ...
            }
            // @audit-issue The getRefundAmounts calculates shortage based on
```

```
            // current balance. Since balance decreases every iteration,
            // the shortage calculation includes amounts from previous legs.
            // Adding this to refundAmounts causes double counting.
            refundAmounts = riskEngine().getRefundAmounts(
                owner, LeftRightSigned.wrap(0), twapTick, ct0, ct1
            ).add(refundAmounts);
            // ...
        }
        unchecked {
            ++leg;
        }
    }
    // ...
}
```

The primary expectation of an option seller calling `_settleLongPremium` is to settle unpaid premiums for long legs efficiently. However, the double-counting mechanism in the loop creates several adverse effects:

1. **Inflated Swap Request**: The cumulative error forces the caller (the seller) to provide more tokens than required to cover the actual shortage;
2. **Forced Exchange**: `getRefundAmounts` operates as a substitution mechanism. When the caller provides the shortage for one token, the system attempts to take an equivalent value of the other token from the option buyer (the `owner`) to refund the caller. Due to the inflated shortage calculation, the option buyer is forced to swap a disproportionately large amount of their collateral to reimburse the caller;
3. **Denial of Service (Revert)**: If the option buyer does not have enough collateral in the second token to cover this artificially inflated refund swap, the transaction will revert. This effectively prevents the option seller from settling the premium and collecting what they are owed, even if the buyer is solvent enough to cover the actual debt;

Recommendation(s): Consider refactoring the logic to accumulate the total realized premium for all legs first, and then calculate the refund amount once after the loop based on the final total debt, rather than accumulating it iteratively.

Status: Fixed.

Update from the client: See correct implementation here. Note: this is a new implementation of the `settleLongPremium` and `getRefundAmounts` is called after all settlements.

6.6 [Low] Attackers can force exercise 0-width long positions with minimal fees to increase victim's `netBorrows`

File(s): `contracts/PanopticPool.sol`

Description: The `PanopticPool` contract allows users to force exercise positions held by other users via the `dispatchFrom(...)` function. This mechanism is primarily intended to clean up out-of-range positions or manage pool health. The cost of this action is calculated by the `exerciseCost(...)` function in the `RiskEngine` contract, which is designed to be expensive for in-range positions and cheaper for out-of-range ones.

However, the `exerciseCost(...)` function skips calculations for legs with a width of 0. If a user holds a position consisting solely of 0-width long legs (which act as "Credit" positions or effectively lending assets to the protocol), the system fails to assign a proper exercise cost.

In `RiskEngine.sol`:

```
function exerciseCost(
    int24 currentTick,
    int24 oracleTick,
    TokenId tokenId,
    PositionBalance positionBalance
) external view returns (LeftRightSigned exerciseFees) {
    // ...
    unchecked {
        bool hasLegsInRange;
        for (uint256 leg = 0; leg < tokenId.countLegs(); ++leg) {
            // ...
            // @audit The function explicitly skips legs with width 0
            if (tokenId.width(leg) == 0) continue;
            // ... logic to determine if leg is in range ...
        }

        // @audit If all legs are width 0, hasLegsInRange remains false
        // The fee defaults to the minimum ONE_BPS (1 basis point)
        int256 fee = hasLegsInRange ?
            -int256(FORCE_EXERCISE_COST) : -int256(ONE_BPS);

        // ...
    }
}
```

The `dispatchFrom(...)` function in `PanopticPool` validates that the position has long legs, but does not validate that those legs have a non-zero width.

```
// contracts/PanopticPool.sol

// ...
} else if (toLength == (finalLength + 1)) {
    // final is one element shorter, that's a force exercise
    // @audit Checks for long legs, but not their width
    if (tokenId.countLongs() == 0) revert Errors.NoLegsExercisable();
    _forceExercise(account, tokenId, twapTick, currentTick);
// ...
```

This creates an attack vector where an attacker can force close a victim's 0-width long position for a trivial cost (`ONE_BPS`).

0-width long positions effectively reduce a user's `netBorrows`. By force exercising (burning) these positions, the attacker increases the victim's `netBorrows`. This can instantly flip a user from a net lender (earning or neutral interest) to a net borrower, forcing them to pay interest on debt they did not intend to hold active.

Recommendation(s): Consider modifying `RiskEngine.sol` to either revert or apply the maximum `FORCE_EXERCISE_COST` when encountering 0-width long legs, as these positions are typically not intended to be forced out of the system.

Status: Fixed.

Update from the client: See correct code here: skips width==0 legs and don't include those into the `longAmounts` used to compute the fee. Also added a `validateIsExercisable` condition here, which ensures that the position to be force exercised has at least a long leg of width !=0.

6.7 [Low] The `exerciseCost(...)` function incorrectly considers in-range positions as out-of-range at the lower tick boundary

File(s): contracts/RiskEngine.sol

Description: The `exerciseCost(...)` function is utilized to determine the settlement fees during a forced exercise, typically calculating the amount the force exercisor must pay to the exercisee. This mechanism ensures that the exercisee is compensated for having their position closed against their will. The cost calculation logic differentiates between in-range and out-of-range positions, where exercising an in-range position incurs a significantly higher cost compared to the minimal base cost of `1 bps` for out-of-range positions.

However, the logic used to determine if a position leg is in-range fails to account for the inclusive lower bound nature of Uniswap V3 liquidity ranges. Uniswap V3 treats a position as active (providing liquidity) in the range `[tickLower, tickUpper)`. If the `currentTick` is exactly equal to `tickLower`, the position is considered in-range.

This issue specifically impacts pools with even tick spacings, which includes the majority of Uniswap V3's standard fee tiers:

- 0.05% fee tier (`tickSpacing` of 10);
- 0.3% fee tier (`tickSpacing` of 60);
- 1% fee tier (`tickSpacing` of 200);

The `exerciseCost(...)` function calculates the distance from the strike price to the range bounds and checks if the `currentTick` falls strictly within this distance.

```
function exerciseCost(...) external view returns (LeftRightSigned exerciseFees) {
    // ...
    for (uint256 leg = 0; leg < tokenId.countLegs(); ++leg) {
        // ...
        {
            int24 range = int24(
                int256(
                    Math.unsafeDivRoundingUp(
                        uint24(
                            tokenId.width(leg) * tokenId.tickSpacing()
                        ),
                        2
                    )
                )
            );
            // ...
            // @audit-issue The strict inequality excludes the lower bound tick
            // from being considered in-range.
            if (Math.abs(currentTick - tokenId.strike(leg)) < range) {
                hasLegsInRange = true;
            }
        }
        // ...
    }
    // ...
}
```

Consider the following scenario for a 0.05% fee tier pool (`tickSpacing` of 10):

1. A position is created with `Strike = 1000` and `Width = 2`;
2. The calculated range delta is `ceil((2 * 10) / 2) = 10`;
3. The position's liquidity range on Uniswap is `[990, 1010)`;
4. If the `currentTick` is `990`, the position is active and in-range on Uniswap;
5. The `exerciseCost(...)` function calculates the absolute distance: `abs(990 - 1000) = 10`;
6. The check `10 < 10` evaluates to `false`;

As a result, `hasLegsInRange` remains `false` even though the position is providing liquidity. The force exercise cost is calculated using the minimal `1 bps` fee instead of the higher fee intended for in-range positions.

This discrepancy means that a user can be force exercised at the lower tick boundary of their position for a significantly lower cost than intended, resulting in lower compensation for the loss of their position.

Recommendation(s): Consider adjusting the range check logic to be inclusive of the lower bound, aligning it with Uniswap V3's liquidity range definition.

Status: Fixed.

Update from the client: See the correct implementation here.

6.8 [Info] The SFPM's registerTokenTransfer(...) prevents transferring 0-width positions due to empty liquidity state

File(s): contracts/SemiFungiblePositionManager.sol

Description: The SemiFungiblePositionManager implements safeTransferFrom and safeBatchTransferFrom, which invoke registerTokenTransfer to move the underlying Uniswap liquidity associated with a position from the sender to the recipient.

The registerTokenTransfer function calculates the expected liquidity for each leg of the transferred token using PanopticMath.getLiquidityChunk and verifies that the sender's stored s_accountLiquidity matches this amount.

```
function registerTokenTransfer(address from, address to, TokenId id, uint256 amount) internal {
    IUniswapV3Pool univ3pool = s_poolIdToPoolData[id.poolId()].pool();
    uint256 numLegs = id.countLegs();
    for (uint256 leg = 0; leg < numLegs; ) {
        // ...
        // @audit Calculates expected liquidity for the transfer amount
        LiquidityChunk liquidityChunk = PanopticMath.getLiquidityChunk(
            id,
            leg,
            uint128(amount)
        );
        // ...
        // @audit Reverts if stored liquidity (fromLiq) does not match expected liquidity
        if (
            LeftRightUnsigned.unwrap(s_accountLiquidity[positionKey_to]) != 0 ||
            LeftRightUnsigned.unwrap(fromLiq) != liquidityChunk.liquidity()
        ) revert Errors.TransferFailed(address(this), from, amount, 0);
        // ...
    }
}
```

However, for positions with 0-width legs (which represent swaps or loans rather than liquidity positions), the minting logic in _createPositionInAMM executes swaps directly and does not update s_accountLiquidity (which remains 0).

If PanopticMath.getLiquidityChunk returns a non-zero liquidity value for the given amount, the validation check in registerTokenTransfer will fail because fromLiq is 0, causing the transfer to revert.

Since the PanopticPool contract is the sole minter and manager of these positions and does not expose transfer functionality to users, this logic is currently dead code. However, it prevents any future use case involving the transfer of 0-width positions.

Recommendation(s): Consider removing the transfer functionality.

Status: Fixed.

Update from the client: Transfers are now disabled, see here.

6.9 [Best Practices] Rounding directions during bonus calculations in `getLiquidationBonus(...)` favor the liquidator

File(s): contracts/RiskEngine.sol

Description: The `getLiquidationBonus(...)` function in the `RiskEngine` contract calculates the collateral bonus paid to a liquidator and determines the resulting protocol loss. When a liquidatee is insolvent in one token but has a surplus in the other, the function attempts to use the surplus to mitigate the shortage by converting the surplus value and adjusting the bonuses accordingly.

However, when covering a shortage in `token0` using a surplus in `token1`, the code subtracts the converted value from `bonus0` using `PanopticMath.convert1to0(...)`. This function defaults to rounding down. Since this value is being subtracted (`bonus0 -= ...`), rounding down the subtrahend results in a smaller deduction from the bonus. This effectively makes the final bonus amount more favorable to the liquidator than to the protocol. The same pattern occurs in the inverse scenario when `token0` surplus is used to cover `token1` shortage.

Standard DeFi security best practices dictate that rounding should generally be performed in favor of the protocol to prevent dust leakage and ensure value is fully conserved.

```
function getLiquidationBonus(
    // ...
) external pure returns (LeftRightSigned, LeftRightSigned) {
    // ...
    unchecked {
        // ...
        // @audit The logic checks if we can cover the shortfall using the other token.
        if (!(paid0 > balance0 && paid1 > balance1)) {
            // liquidatee cannot pay back the liquidator fully in either token...
            if ((paid0 > balance0)) {
                // ...
                bonus1 += Math.min(
                    balance1 - paid1,
                    PanopticMath.convert0to1(
                        paid0 - balance0, atSqrtPriceX96
                    )
                );
                // @audit-issue The `convert1to0` rounds down. Since we are subtracting,
                // a smaller value favors the liquidator.
                bonus0 -= Math.min(
                    PanopticMath.convert1to0(
                        balance1 - paid1, atSqrtPriceX96
                    ),
                    paid0 - balance0
                );
            }
            if ((paid1 > balance1)) {
                // ...
                bonus0 += Math.min(
                    balance0 - paid0,
                    PanopticMath.convert1to0(
                        paid1 - balance1, atSqrtPriceX96
                    )
                );
                // @audit-issue Similar issue here with `convert0to1` rounding down
                // while being subtracted.
                bonus1 -= Math.min(
                    PanopticMath.convert0to1(
                        balance0 - paid0, atSqrtPriceX96
                    ),
                    paid1 - balance1
                );
            }
        }
        // ...
    }
}
```

Recommendation(s): Consider using `PanopticMath.convert1to0RoundingUp(...)` and `PanopticMath.convert0to1RoundingUp(...)` when calculating the amount to subtract from the bonus to ensure rounding favors the protocol.

Status: Fixed.

Update from the client: See code here.

6.10 [Best Practices] The MAX_TICK_DELTA_SUBSTITUTION constant is defined but not used

File(s): contracts/PanopticPool.sol

Description: The PanopticPool contract defines the constant MAX_TICK_DELTA_SUBSTITUTION. According to the inline documentation, this constant is intended to ensure that token substitutions during _forceExercise(...) and _settleLongPremium(...) operations occur safely at a price close to the market, with a maximum allowed delta of approximately 2%.

However, this constant is not referenced or used anywhere within the contract's logic. The dispatchFrom(...) function, which handles the dispatching of force exercises and premium settlements, does not implement any check utilizing this value.

```
// ...
/// @notice The maximum allowed delta (~2%) between the latestTick and the medianTick/spotTick during
 ↪  forceExercise/settleLongPremium.
/// @dev Ensures token substitution between two accounts is settled safely at a price close to market.
int256 internal constant MAX_TICK_DELTA_SUBSTITUTION = 203;
// ...
```

This means that the documented safety check for price deviation during substitutions is currently absent or the constant is redundant if the logic was intentionally removed.

Recommendation(s): Consider reviewing the intended logic for force exercises and long premium settlements. If the check is no longer needed, consider removing the unused constant and its associated comments to clean up the codebase.

Status: Fixed.

Update from the client: Fixed.

6.11 [Best Practices] The `_burn(...)` function uses an unchecked block for `_internalSupply` decrement which could silently underflow

File(s): contracts/PanopticPool.sol

Description: The `ERC20Minimal` contract is an abstract ERC20 implementation inherited by `CollateralTracker`, which functions as an ERC4626-style vault for managing collateral shares. Users deposit underlying assets and receive shares representing their proportional ownership of the vault's assets. The `_burn(...)` function is used to redeem shares when users withdraw assets, settle accrued interest, or close option positions. However, the function decrements `_internalSupply` inside an unchecked block, assuming that a user's balance can never exceed the total supply:

```
function _burn(address from, uint256 amount) internal {
    balanceOf[from] -= amount;
    // Cannot underflow because a user's balance
    // will never be larger than the total supply.
    unchecked {
        // @audit The assumption does not hold in Panoptic; this can underflow.
        _internalSupply -= amount;
    }

    emit Transfer(from, address(0), amount);
}

```

The assumption that `balanceOf[user] <= _internalSupply` does not hold in the broader context of Panoptic protocol. The `CollateralTracker` contract, which inherits from `ERC20Minimal`, directly manipulates user balances via the `delegate(...)` and `revoke(...)` functions. These functions inflate and deflate a user's balance by `type(uint248).max` to facilitate solvency checks during liquidations and exercises, but they do not update the `_internalSupply`.

```
function delegate(address delegatee) external onlyPanopticPool {
    // @audit Balance is inflated directly without updating _internalSupply
    balanceOf[delegatee] += type(uint248).max;
}

function revoke(address delegatee) external onlyPanopticPool {
    // @audit Balance is deflated directly without updating _internalSupply
    balanceOf[delegatee] -= type(uint248).max;
}

```

This means that during specific operations (like `dispatchFrom(...)`), a user's balance is temporarily decoupled from the supply as force exercise, settle long premium, and liquidation flows inflate user balances. Since these flows manipulate balances independently of `_internalSupply`, there may be a possibility to reach a state where a user can burn more than the total supply, causing `_internalSupply` to silently underflow. As a matter of defensive programming, `_internalSupply` should never go below zero. If it does, the transaction should revert rather than continue with corrupted state.

Recommendation(s): Consider removing the unchecked block from the `_internalSupply` decrement in `_burn(...)`. While this introduces a small gas overhead, it ensures that any unexpected underflow will revert the transaction rather than silently corrupt the supply accounting.

Status: Fixed.

Update from the client: Also removed the unchecked for mint, see code here.

6.12 [Best Practices] The `dispatchFrom(...)` function contains an unused local variable

File(s): `contracts/PanopticPool.sol`

Description: In the `dispatchFrom(...)` function in `PanopticPool.sol`, the local variable `LeftRightSigned exchangedAmounts` captures the return value from the `_liquidate(...)` function call. The `_liquidate` function returns `bonusAmounts`, which represents the liquidation bonus calculated for the liquidator.

```
function dispatchFrom(...) external {
    // ...
    // @audit-issue This variable is unused.
    LeftRightSigned exchangedAmounts;
    {
        // ...
        if (solvent == numberOfTicks) {
            // ...
        } else if (solvent == 0) {
            // ...
            // @audit The return value is assigned here but never used afterwards.
            exchangedAmounts = _liquidate(account, positionIdListTo, twapTick, currentTick);
        } else {
            revert Errors.NotMarginCalled();
        }
    }
    // ...
}
```

However, this `exchangedAmounts` variable is never used in the subsequent logic of `dispatchFrom`. This makes the return value assignment in `dispatchFrom` redundant. While this doesn't affect functionality (since all necessary state changes and events occur within `_liquidate`), it clutters the code.

Recommendation(s): Consider removing the unused variable assignment or documenting why the return value is intentionally captured but unused.

Status: Fixed.

Update from the client: Fixed.

6.13 [Best Practices] The `validateIsExercisable(...)` function is unused and contains outdated logic

File(s): contracts/PanopticPool.sol

Description: The `TokenId` library defines the `validateIsExercisable(...)` function, which is currently unused in the main protocol contracts. The function's logic and documentation imply that a position must have at least one long leg that is "far-out-of-the-money" (outside its price range) to be exercisable.

```
/// @notice Validate that a position `self` and its legs/chunks are exercisable.
/// @dev At least one long leg must be far-out-of-the-money (i.e. price is outside its range).
// ...
function validateIsExercisable(...) internal pure {
    unchecked {
        uint256 numLegs = self.countLegs();
        for (uint256 i = 0; i < numLegs; ++i) {
            // ...
            // @audit Check if the price is outside this chunk.
            if ((currentTick >= _strike + rangeUp) || (currentTick < _strike - rangeDown)) {
                // @audit If this leg is long and the price is beyond the leg's range, the position is valid.
                if (self.isLong(i) == 1) return;
            }
        }
    }

    // @audit Fail if position has no legs that is far-out-of-the-money.
    revert Errors.NoLegsExercisable();
}
```

However, the actual force exercise logic implemented in the `dispatchFrom(...)` function of `PanopticPool` only validates that the position contains at least one long leg, regardless of whether it is in-the-money or out-of-the-money

```
// ...
// @audit The current implementation only checks for the existence of long legs.
if (tokenId.countLongs() == 0) revert Errors.NoLegsExercisable();
_forceExercise(account, tokenId, twapTick, currentTick);
// ...
```

This discrepancy suggests that `validateIsExercisable(...)` is dead code containing outdated logic that contradicts the current protocol design for force exercises.

Recommendation(s): Consider removing the unused `validateIsExercisable(...)` function from the `TokenId` library. This will reduce code complexity and eliminate confusion regarding the requirements for force exercises.

Status: Fixed.

Update from the client: The function has been revived when addressing the *[Low] Attackers can force exercise 0-width...* issue: we now use the `validateIsExercisable` condition here, which ensures that the position to be force exercised has at least a long leg of width !=0.

6.14 [Best Practices] The overloaded `withdraw(...)` function in `CollateralTracker` bypasses the minimum asset retention mechanism

File(s): `contracts/CollateralTracker.sol`

Description: The contract `CollateralTracker` implements two variations of the `withdraw(...)` function to handle asset redemptions. The standard ERC4626 implementation utilizes the `maxWithdraw(...)` function to determine the limit of assets a user can withdraw.

The `maxWithdraw(...)` function includes logic to retain at least `1` unit of assets in the contract (`s_depositedAssets - 1`), preventing the pool from being completely emptied. This is used to maintain invariants related to share pricing. The standard `withdraw(...)` function enforces this limit:

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner
) external returns (uint256 shares) {
    _accrueInterest(owner, IS_NOT_DEPOSIT);
    // @audit Enforces the retention of 1 unit of asset
    if (assets > maxWithdraw(owner)) revert Errors.ExceedsMaximumRedemption();
    if (assets == 0) revert Errors.BelowMinimumRedemption();
    // ...
}
```

However, the overloaded `withdraw(...)` function, which is designed to handle withdrawals for users with open positions, does not include this check. It calculates the shares to burn and processes the transfer without verifying if the withdrawal amount exceeds the `s_depositedAssets - 1` threshold.

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner,
    TokenId[] calldata positionIdList,
    bool usePremiaAsCollateral
) external returns (uint256 shares) {
    _accrueInterest(owner, IS_NOT_DEPOSIT);
    // @audit-issue Skips the maxWithdraw check, allowing full withdrawal
    shares = previewWithdraw(assets);
    if (assets == 0) revert Errors.BelowMinimumRedemption();
    // ...
}
```

Crucially, this overloaded function can be called by a user who does not have any open positions. If the caller provides an empty `positionIdList`, the `validateCollateralWithdrawable(...)` call in the `PanopticPool` contract will skip the solvency check entirely.

This inconsistency allows a user to potentially withdraw the entire balance of deposited assets using the overloaded function, breaking the invariant established by the standard implementation. If the `totalAssets()` drops to zero, it will cause `previewDeposit(...)` to revert due to division by zero, effectively bricking subsequent deposits into the `CollateralTracker`.

Recommendation(s): Consider implementing consistent handling of the maximum withdrawable amount across both withdrawal functions to ensure the minimum asset retention invariant is maintained.

Status: Fixed.

Update from the client: See code here.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Panoptic's documentation

The Panoptic team has provided exceptional documentation, both through their website and extensive inline code comments. Complex mechanisms and edge cases are explained in great detail, providing the reasoning behind design choices. The codebase notably features ASCII diagrams that effectively visualize abstract concepts. Furthermore, the client was highly collaborative, offering extremely detailed walkthroughs of the codebase and answering inquiries during regular synchronization calls with the Nethermind Security team.

8 Test Suite Evaluation

Remarks about Panoptic's test suite

The Panoptic team has developed an extensive test suite comprising unit tests, fuzz tests, and end-to-end invariant tests covering the entire protocol. Math-heavy functions are thoroughly tested, and the test cases are well-structured, aiding auditors in understanding the protocol's intended use cases. However, a potential area for improvement is expanding test coverage for newly introduced features, such as 0-width position handling and the liquidation flow, specifically focusing on liquidator bonuses and refunding mechanisms, where issues were identified.

8.1 Tests Coverage

```
> forge coverage
Ran 22 test suites in 590.87s (2588.78s CPU time): 516 tests passed, 10 failed, 2 skipped (528 total tests)
```

Coverage table adapted from the output of `forge coverage --report summary --allow-failure`. Metrics include the core protocol contracts, excluding test harnesses and scripts.

File	% Lines	% Statements	% Branches	% Funcs
Core Protocol				
contracts/CollateralTracker.sol	98.11%	99.39%	95.74%	93.55%
contracts/PanopticFactory.sol	100.00%	100.00%	100.00%	100.00%
contracts/PanopticPool.sol	98.31%	97.70%	79.63%	94.74%
contracts/RiskEngine.sol	100.00%	100.00%	100.00%	100.00%
contracts/SemiFungiblePositionManager.sol	100.00%	100.00%	100.00%	100.00%
Base Contracts				
contracts/base/FactoryNFT.sol	29.45%	29.45%	5.50%	90.00%
contracts/base/MetadataStore.sol	100.00%	100.00%	100.00%	100.00%
contracts/base/Multicall.sol	0.00%	0.00%	0.00%	0.00%
Libraries				
contracts/libraries/CallbackLib.sol	100.00%	100.00%	100.00%	100.00%
contracts/libraries/EfficientHash.sol	17.39%	15.79%	100.00%	25.00%
contracts/libraries/FeesCalc.sol	100.00%	100.00%	100.00%	100.00%
contracts/libraries/InteractionHelper.sol	87.50%	93.33%	50.00%	100.00%
contracts/libraries/Math.sol	92.89%	93.06%	87.80%	100.00%
contracts/libraries/PanopticMath.sol	97.35%	97.89%	92.16%	100.00%
contracts/libraries/SafeTransferLib.sol	64.29%	58.62%	25.00%	66.67%
Tokens				
contracts/tokens/ERC1155Minimal.sol	82.61%	84.09%	37.50%	100.00%
contracts/tokens/ERC20Minimal.sol	100.00%	100.00%	100.00%	100.00%
Types				
contracts/types/LeftRight.sol	100.00%	100.00%	100.00%	100.00%
contracts/types/LiquidityChunk.sol	100.00%	100.00%	100.00%	100.00%
contracts/types/Pointer.sol	100.00%	100.00%	100.00%	100.00%
contracts/types/PoolData.sol	100.00%	100.00%	100.00%	100.00%
contracts/types/PositionBalance.sol	74.19%	68.18%	100.00%	84.62%
contracts/types/TokenId.sol	97.85%	99.04%	91.67%	100.00%
Global Average (All Files)	**83.86%**	**85.38%**	**67.98%**	**82.61%**

8.2 Automated Tools

8.2.1 AuditAgent

The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.