Report

v. 4.0

Customer

Panoptic

## Smart Contract Audit

# Panoptic

26th April 2023

Report prepared by

ABDK Consulting

# Contents

# 1 Changelog

| # | Date | Author | Description |
|---|---|---|---|
| 0.1 | 11.04.23 | A. Zveryanskaya | Initial Draft |
| 0.2 | 11.04.23 | A. Zveryanskaya | Minor revision |
| 1.0 | 11.04.23 | A. Zveryanskaya | Release |
| 1.1 | 23.04.23 | A. Zveryanskaya | CVF-5, 6 are downgraded to Major |
| 1.2 | 23.04.23 | A. Zveryanskaya | CVF-7, 9, 10, 11, 18, 19, 23, 46 are downgraded to Moderate |
| 1.3 | 23.04.23 | A. Zveryanskaya | CVF-109, 110 are down-graded to Minor |
| 1.4 | 23.04.23 | A. Zveryanskaya | CVF-8, 13, 30, 35, 41, 44, 280-299 are removed |
| 1.5 | 23.04.23 | A. Zveryanskaya | CVF-31, 39, 139, 160, 176, 190, 205 are marked as Fixed |
| 1.6 | 23.04.23 | A. Zveryanskaya | CVF-161 missed code block is added |
| 2.0 | 24.04.23 | A. Zveryanskaya | Release |
| 2.1 | 24.04.23 | A. Zveryanskaya | CVF-53, 60, 61, 74, 119, 207, 214, 223 are marked as Fixed |
| 2.2 | 24.04.23 | A. Zveryanskaya | CVF-75, 106, 125, 165 typos are fixed |
| 3.0 | 24.04.23 | A. Zveryanskaya | Release |
| 3.1 | 26.04.23 | A. Zveryanskaya | CVF-7, 52, 112-117, 120-129, 264-279 are removed |
| 4.0 | 26.04.23 | A. Zveryanskaya | Release |

# 2 Introduction



The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

Panoptic is a perpetual, oracle-free, instant-settlement options trading protocol on the Ethereum blockchain. Panoptic enables the permissionless trading of options on top of any asset pool in the Uniswap v3 ecosystem and seeks to develop a trustless, permissionless, and composable options product, i.e., do for decentralized options markets what x·y=k automated market maker protocols did for spot trading.

# 3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Files:

| | | | |
|---|---|---|---|
| **/** | | | |
| | CollateralTracker.sol | PanopticFactory.sol | PanopticPool.sol |
| | SemiFungiblePosition Manager.sol | | |
| **libraries/** | | | |
| | Errors.sol | FeesCalc.sol | LeftRight.sol |
| | LiquidityChunk.sol | Math.sol | PanopticMath.sol |
| | TickPriceFeeInfo.sol | TokenId.sol | |
| **uniswapv3_periphery/base/** | | | |
| | PeripheryPayments.sol | | |

# 4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Minor issues** contain code style, best practices and other recommendations.

# 5 Our findings

We found 4 critical, 32 major, and a few less important issues. All identified Critical and Major issues have been fixed or otherwise addressed in collaboration with the client.

| Issues | Active | Fixed |
|---|---|---|
| Severity: Critical | 0 | 4 |

| Major | Active | Fixed |
|---|---|---|
| | 1 | 31 |

| Moderate | Active | Fixed |
|---|---|---|
| | 44 | 21 |

| Minor | Active | Fixed |
|---|---|---|
| | 54 | 85 |

Fixed 141 out of 240 issues

# 6 Critical Issues

## CVF-1. FIXED

- **Category** Flaw
- **Source** PeripheryPayments.sol

**Recommendation** This condition ignores the “payer” argument. Should include “&& payer == address(this)”.

**Client Comment** *We no longer use this library.*

```
22 if (token == WETH9 && address(this).balance >= value) {
```

## CVF-2. FIXED

- **Category** Flaw
- **Source** CollateralTracker.sol

**Description** Despite the comment, a negative swapped amount is actually replaced with its absolute value rather than zero.

**Client Comment** *Updated the logic as described by the code comment.*

```
1169 // set swapped amount to zero if it is negative
```

```
1171 swappedAmount = swappedAmount < 0 ? -swappedAmount : swappedAmount;
```

## CVF-3. FIXED

- **Category** Unclear behavior
- **Source** SemiFungiblePositionManager.sol

**Description** It seems the caller doesn’t use this information. It still uses the whole liquidity chunk as if all the requested liquidity were available.

**Client Comment** *This condition is now enforced here.*

```
614 // note also later in this fct that we tell the caller "you can only
    ↪  move out startingLiquidity"
```

## CVF-4. FIXED

- **Category** Unclear behavior
- **Source** SemiFungiblePositionManager.sol

**Description** In case the starting liquidity is less than liquidity chunk liquidity, the liquidity chunk amounts should be decreased accordingly.

**Client Comment** *We decided to enforce this and have the function revert in this situation.*

```
731 uint256 liquidityChunk,
```

```
733 uint256 liquidities,
```

# 7 Major Issues

## CVF-5. FIXED

- **Category** Flaw
- **Source** TokenId.sol

**Description** The ”MAX_LEG_WIDTH” special value for ”self.width(legIndex)” may cause the range check above to fail.

**Recommendation** Consider performing the range check only after the special value check.

**Client Comment** *The range check was moved to the end.*

```
401 (legLowerTick, legUpperTick) = self.width(legIndex) == MAX_LEG_WIDTH
```

## CVF-6. INFO

- **Category** Flaw
- **Source** LeftRight.sol

**Description** This logic cannot correctly handle negative ”right” values. For example: Input: self = 0×2 (0, 2), right = -1 Expected output: 0×1 (0, 1) Actual output: 0×100000000000000000000000000000001 (1, 1)

**Client Comment** *This function is not intended to handle existing bits in the slot it’s writing to. We have ’add’ and ’sub’ functions for this purpose.*

```
94 return self + (int256(right) & RIGHT_HALF_BIT_MASK);
```

## CVF-12. FIXED

- **Category** Flaw
- **Source** PanopticPool.sol

**Description** This code runs out of gas if the list is longer than 256 entries.

**Recommendation** Consider handling this situation explicitly.

**Client Comment** *We have changed the type of the iterator to a uint256.*

```
190 for (uint8 i; i < positionIdList.length; ) {
        // store user balance in the array to be returned
        balances[i] = _getUserOptionsBalance(user, positionIdList[i]);
            ↪ // get balance from storage
        unchecked {
            ++i;
        }
    }
```

## CVF-14. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** The expression “mintTokenId.countLegs()” is calculated on every loop iteration.

**Recommendation** Consider calculating once before the loop.

```
767 for (uint256 index = 0; index < mintTokenId.countLegs(); ) {
```

## CVF-15. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** The expression “burnTokenId.countLegs()” is calculated on every loop iteration.

**Recommendation** Consider calculating once before the loop.

```
1010 for (uint256 i = 0; i < burnTokenId.countLegs(); ) {
```

## CVF-16. FIXED

- **Category** Unclear behavior
- **Source** PanopticPool.sol

**Description** It is unclear what negative values for this argument mean.

**Recommendation** Consider either forbidding negative values or explaining their semantics.

**Client Comment** *We have changed the type to a uint to make clear that the argument cannot be negative.*

1376
```
int128 requestedAmount
```

## CVF-17. FIXED

- **Category** Unclear behavior
- **Source** PanopticPool.sol

**Description** The comment tells about “administrating” while the error tells about “liquidating” which is not the same. The function “_administrateAccount” is called not only during liquidation, but also during forced exercising.

**Recommendation** Consider rephrasing the comment or moving this this check into the calling functions.

**Client Comment** *This check was removed altogether.*

1484
```
// cannot force administrate own account
if (_msgSender() == account) revert Errors.CannotSelfLiquidate();
```

## CVF-20. FIXED

- **Category** Procedural
- **Source** CollateralTracker.sol

**Description** Here the Uniswap pool address is implicitly passed to the collateral tracker. Such implicit data flows are error prone and make code harder to read.

**Recommendation** Consider passing the Uniswap pool address via an explicit argument.

148
```
IUniswapV3Pool uniswapPool = IPanopticPool(_msgSender()).univ3pool()
    ↪ ;
```

## CVF-21. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The token addresses are queried by the Uniswap pool several times, which is gas consuming.

**Recommendation** Consider querying once and reusing.

```
152     !(underlyingAddress == uniswapPool.token0() ||
            underlyingAddress == uniswapPool.token1())
```

```
157 if ((uniswapPool.token0() == address(0)) || (uniswapPool.token1() ==
    ↪  address(0)))
```

```
168 s_univ3token0 = uniswapPool.token0();
    s_univ3token1 = uniswapPool.token1();
```

```
172 s_underlyingIsToken0 = address(underlyingAddress) == uniswapPool.
    ↪ token0();
```

## CVF-22. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** Here values are checked for validity after writing them to the storage. This looks like waste of gas.

**Recommendation** Consider performing all validity checks before writing anything into the storage.

**Client Comment** *This whole section was refactored such that the parameters are put into a struct and written without explicit, redundant checks.*

```
228     ((s_MAINTENANCE_MARGIN_RATIO = int256(int16(uint16(parameterData
           ↪ )))) <= 0) ||
        ((s_COMMISSION_FEE_MIN = int128(int16(uint16(parameterData >>
           ↪ 16)))) <= 0) ||
230     ((s_COMMISSION_FEE_MAX = int128(int16(uint16(parameterData >>
           ↪ 32)))) <= 0) ||
        ((s_COMMISSION_START_UTILIZATION = int128(int16(uint16(
           ↪ parameterData >> 48)))) <= 0) ||
        ((s_SELL_COLLATERAL_RATIO = int128(int16(uint16(parameterData >>
           ↪  64)))) <= 0) ||
        ((s_BUY_COLLATERAL_RATIO = int128(int16(uint16(parameterData >>
           ↪ 80)))) <= 0) ||
```

```
235     ((s_TARGET_POOL_UTILIZATION = int128(int16(uint16(parameterData
           ↪ >> 128)))) <= 0) ||
        ((s_SATURATED_POOL_UTILIZATION = int128(int16(uint16(
           ↪ parameterData >> 144)))) <= 0)
```

```
240 if (s_EXERCISE_COST >= 0) revert Errors.InvalidInputParameters();
```

## CVF-24. FIXED

- **Category** Documentation
- **Source** CollateralTracker.sol

**Description** There is no such logic in the function.

**Client Comment** *The comment was clarified to reflect the current behavior.*

```
282 * @dev Function that adds/removes amount from locked and inAMM
       ↪ storage, will lock all funds if there is <100 wei (dust
       ↪ threshold)
```

## CVF-25. FIXED

- **Category** Unclear behavior
- **Source** CollateralTracker.sol

**Description** Here multiplication after division is performed.

**Recommendation** Consider dividing once at the end of calculation.

**Client Comment** *We moved away from the MulDiv here and now perform all the multiplications first.*

375
```
uint256 valueRatio1 = FullMath.mulDiv(
    FullMath.mulDiv(tokenData.rightSlot(), FixedPoint96.Q96,
       ↪ sqrtPriceX96),
    DECIMALS,
    tokenValue
```

## CVF-26. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The expression ”tokenId.countLegs()” is calculated on every loop iteration.

472
```
for (uint256 index = 0; index < tokenId.countLegs(); ) {
```

## CVF-27. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This check seems redundant.

**Recommendation** Consider removing it or explaining why it is necessary.

**Client Comment** *We removed the check.*

759
```
require(balanceOf(_user) <= balanceBefore + shares);
```

## CVF-28. FIXED

- **Category** Unclear behavior
- **Source** CollateralTracker.sol

**Description** The actual amount of shares burned by this function could be less than this value.

**Recommendation** Consider returning the actual amount of shares burned.

**Client Comment** *This issue is no longer relevant because we return assets when shares are specified and vice versa, conforming to the ERC4626 standard.*

766
```
* @param shares Amount of shares to be withdrawn
```

## CVF-29. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This check seems redundant.

**Recommendation** Consider removing it or explaining why it is necessary.

**Client Comment** *We removed the check.*

807
```
require(balanceOf(_user) <= balanceBefore);
```

## CVF-31. FIXED

- **Category** Flaw
- **Source** CollateralTracker.sol

**Description** The actual check doesn't guarantee that the list passed as an argument matches the actual user positions.

**Client Comment** *This code is no longer present in the codebase, as we now rely solely on computing a hash of the positions and comparing it to a stored value.*

1237
```
// ensure that the incoming list of active positions matches what
   ↪ the account has
```

1239
```
if (positionIdList.length == 0 && numActivePositions > 0) revert
   ↪ Errors.InputListFail();
```

## CVF-32. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The expression ”tokenId.countLegs()” is calculated on every loop iteration.

**Recommendation** Consider calculating once and reusing.

```
1284 for (uint256 index = 0; index < tokenId.countLegs(); index++) {
```

## CVF-33. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Underflow is possible here.

**Recommendation** Consider using safe conversion.

**Client Comment** *MAINTENANCE_MARGIN_RATIO is now stored as a uint, therefore, conversion is no longer necessary.*

```
1444 uint256(s_MAINTENANCE_MARGIN_RATIO),
```

## CVF-34. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Phantom overflow is possible here, i.e. a situation when the final calculation result would fit into the destination type, while some intermediary calculation overflows.

**Recommendation** Consider using the “muldiv” function or calculating in 256 bits.

**Client Comment** *We modified this to calculate in 256 bits.*

```
1565 required = required.toRightSlot((sellCollateral * int128(amountMoved
       ↪ )) / DECIMALS_128);
```

## CVF-36. FIXED

- **Category** Suboptimal
- **Source** SemiFungiblePositionManager.sol

**Description** The expression "tokenId.countLegs()" is calculated on every loop iteration.

**Recommendation** Consider calculating once before the loop.

469
```
for (uint256 index = 0; index < tokenId.countLegs(); ++index) {
```

## CVF-37. FIXED

- **Category** Suboptimal
- **Source** FeesCalc.sol

**Description** The expression "tokenId.countLegs()" is calculated on every loop iteration.

**Recommendation** Consider calculating once before the loop.

129
```
for (uint256 index = 0; index < tokenId.countLegs(); ) {
```

## CVF-38. FIXED

- **Category** Suboptimal
- **Source** PanopticMath.sol

**Description** The expression "tokenId.countLegs()" is calculated on every loop iteration.

**Recommendation** Consider calculating once before the loop.

152
```
for (uint256 legIndex = 0; legIndex < tokenId.countLegs(); ) {
```

## CVF-39. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** It is possible to deploy two contracts whose addresses don't differ in the lower 80 bits.

**Recommendation** Consider using a more secure approach, such as maintaining a list of valid Uniswap pools and using an index in this list.

**Client Comment** *In order to avoid possible collisions between pool IDs, we've changed the way poolIDs are calculated. In most cases, it will remain the last 8 bytes of the address for legibility. However, in the event of a collision, the poolID will be incremented by 32 bits of the hash keccak256(abi.encodePacked(token0, token1, fee). This is done deliberately so it is not possible to manipulate the pool address and the increment value seperately, as the same values are used as part of the CREATE2 salt when Uniswap pools are created by the factory. If there are multiple collisions, it will be incremented again by the same value until it no longer collides. This ensures that it is not feasible to either:*
*1. Prevent a pool from being deployed*
*2. Overwrite an existing pool with one that posses at colliding poolID*

```
27 * (1) univ3pool        80bits  : first 10 bytes of the Uniswap v3
      ↪ pool address (first 80 bits; little-endian)
```

## CVF-40. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** The value calculated here is not used in case the leg width is "MAX_LEG_WIDTH".

**Recommendation** Consider not performing this calculation in such a case.

```
395 int24 oneSidedRange = (self.width(legIndex) * tickSpacing) / 2;
```

## CVF-42. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** The expression ”self.countLegs()” is calculated on every loop iteration.

**Recommendation** Consider calculating once and reusing.

```
523 for (uint256 i = 0; i < self.countLegs(); ++i) {
```

## CVF-43. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** The expression ”self.countLegs()” is calculated on every loop iteration.

**Recommendation** Consider calculating once and reusing.

```
552 for (uint256 i = 0; i < self.countLegs(); ++i) {
```

## CVF-45. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Recommendation** This check could be optimized using a bit mask that covers all the checked fields.

```
626 (XORtokenId.optionRatio(i) != 0) ||
    (XORtokenId.numeraire(i) != 0) ||
    (XORtokenId.isLong(i) != 0) ||
    (XORtokenId.tokenType(i) != 0) ||
630 (XORtokenId.riskPartner(i) != 0)
```

## CVF-48. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This could be simplified as: unchecked { z = x - y; } require (z <= x); require (uint128 (z) <= uint128 (x));

191
```
unchecked {
    uint128 leftSub = x.leftSlot() - y.leftSlot();
    uint128 rightSub = x.rightSlot() - y.rightSlot();

    if ((leftSub > x.leftSlot()) || (rightSub > x.rightSlot()))
        revert Errors.UnderOverFlow();

    return z.toRightSlot(rightSub).toLeftSlot(leftSub);
}
```

## CVF-49. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This logic could be simplified: unchecked { int256 left = int256(uint256(x.leftSlot())) + y.leftSlot(); int128 left128 = int128(left); require (left128 == left); int256 right = int256(uint256(x.rightSlot())) + y.rightSlot(); int128 right128 = int128(right); require (right128 == right); return z.toRightSlot(right128).toLeftSlot(left128); }

```
246 unchecked {
        if (x.leftSlot() == type(uint128).max || x.rightSlot() == type(
            ↪ uint128).max)
            revert Errors.UnderOverFlow();

250     int128 leftSum = int128(x.leftSlot()) + y.leftSlot();
        int128 rightSum = int128(x.rightSlot()) + y.rightSlot();

        if (
            ((leftSum < int128(x.leftSlot())) && (y.leftSlot() > 0)) ||
            ((leftSum > int128(x.leftSlot())) && (y.leftSlot() < 0)) ||
            ((rightSum < int128(x.rightSlot())) && (y.rightSlot() > 0))
                ↪ ||
            ((rightSum > int128(x.rightSlot())) && (y.rightSlot() < 0))
        ) revert Errors.UnderOverFlow();

260     return z.toRightSlot(rightSum).toLeftSlot(leftSum);
    }
```

## CVF-50. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This logic could be simplified: unchecked { int256 left = int256(x.leftSlot()) + y.leftSlot(); int128 left128 = int128(left); require (left128 == left); int256 right = int256(x.rightSlot()) + y.rightSlot(); int128 right128 = int128(right); require (right128 == right); return z.toRightSlot(right128).toLeftSlot(left128); }

```
271 unchecked {
        int128 leftSum = x.leftSlot() + y.leftSlot();
        int128 rightSum = x.rightSlot() + y.rightSlot();

        if (
            ((leftSum < x.leftSlot()) && (y.leftSlot() > 0)) ||
            ((rightSum < x.rightSlot()) && (y.rightSlot() > 0)) ||
            ((leftSum > x.leftSlot()) && (y.leftSlot() < 0)) ||
            ((rightSum > x.rightSlot()) && (y.rightSlot() < 0))
280     ) revert Errors.UnderOverFlow();

        return z.toRightSlot(rightSum).toLeftSlot(leftSum);
    }
```

## CVF-51. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This logic could be simplified: unchecked { int256 left = int256(x.leftSlot()) - y.leftSlot(); int128 left128 = int128(left); require (left128 == left); int256 right = int256(x.rightSlot()) - y.rightSlot(); int128 right128 = int128(right); require (right128 == right); return z.toRightSlot(right128).toLeftSlot(left128); }

```
293 unchecked {
        int128 leftSub = x.leftSlot() - y.leftSlot();
        int128 rightSub = x.rightSlot() - y.rightSlot();
        if (
            ((leftSub > x.leftSlot()) && (y.leftSlot() > 0)) ||
            ((rightSub > x.rightSlot()) && (y.rightSlot() > 0)) ||
            ((leftSub < x.leftSlot()) && (y.leftSlot() < 0)) ||
300         ((rightSub < x.rightSlot()) && (y.rightSlot() < 0))
        ) revert Errors.UnderOverFlow();

        return z.toRightSlot(rightSub).toLeftSlot(leftSub);
    }
```

# 8 Moderate Issues

## CVF-9. INFO

- **Category** Suboptimal
- **Source** PanopticFactory.sol

**Description** This function constructs a message and calculates the hash of this message. Messages constructed on different loop iterations differ only in one field (salt).

**Recommendation** Consider constructing the message once and only change salt within it.

**Client Comment** *The function this is in: 'minePoolAddress' is never intended to be called on-chain, so gas efficiency is somewhat moot. Rather, it is a convenience function for people who want to mine pool addresses on Etherscan instead of doing the work of installing and configuring software.*

```
228 newPoolAddress = POOL_REFERENCE.predictDeterministicAddress(
```

## CVF-10. INFO

- **Category** Suboptimal
- **Source** PanopticFactory.sol

**Description** This function constructs a message and calculates the hash of this message. Messages constructed on different loop iterations differ only in one field (nonce).

**Recommendation** Consider constructing the message once and only change nonce within it.

**Client Comment** *The function this is in: 'minePoolAddress' is never intended to be called on-chain, so gas efficiency is somewhat moot. Rather, it is a convenience function for people who want to mine pool addresses on Etherscan instead of doing the work of installing and configuring software.*

```
229 _getSalt(v3Pool, deployer, nonce)
```

## CVF-11. INFO

- **Category** Flaw
- **Source** PanopticPool.sol

**Description** The returned value is ignored.

**Recommendation** Consider explicitly requiring the returned value to be true.

**Client Comment** *We cannot productively handle a failure here, so we assume the result to be true. While it's technically possible under the ERC20 standard, in practice, no legitimate token we know of fails upon approval, especially by returning 'false' instead of reverting.*

```
135 IERC20(s_token0).approve(address(sfpm), type(uint256).max);
    IERC20(s_token1).approve(address(sfpm), type(uint256).max);
```

```
139 IERC20(s_token0).approve(address(s_collateralToken0), type(uint256).
       ↪ max);
140 IERC20(s_token1).approve(address(s_collateralToken1), type(uint256).
       ↪ max);
```

## CVF-18. FIXED

- **Category** Bad naming
- **Source** PanopticPool.sol

**Description** These two functions are complimentary getter and setter for the same thing, but are named very differently.

**Recommendation** Consider naming consistently.

**Client Comment** *This was refactored with consistent naming,*

```
1695 function _setOptionsPositionsFingerprint(address user, uint128
       ↪ newFingerprint) private {
```

```
1764 function _getUserPositionHash(address user) private view returns (
       ↪ uint128) {
```

## CVF-19. INFO

- **Category** Procedural
- **Source** CollateralTracker.sol

**Description** This contract uses many different integer types: uint256, int256, uint128, int128, uint64, int64; and often performs unsafe conversions between these types, and performs unchecked calculations in these types. Such approach is very error-prone.

**Recommendation** Consider using a single integer type, for example “int256” everywhere for calculations, and perform checked conversion to narrower type only before writing a value into a storage variable.

**Client Comment** *This is a fair point, but because this a more general suggestion and not a specific issue that has been raised, we have determined that at this time we will not be addressing the issue due to time constraints and an upcoming code freeze. We will keep it in mind moving forward.*

```
43 contract CollateralTracker is ERC20 {
```

## CVF-23. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The current underlying token balance of the Panoptic pool is obtained twice: once here and another time inside the ”_totalBalance” function.

**Recommendation** Consider refactoring to obtain the balance once.

**Client Comment** *We chose not to implement this suggestion for readability, since this function is primarily intended for offchain queries.*

```
271 balance = IERC20(s_underlyingToken).balanceOf(address(s_panopticPool
    ↪ ));
currentTotalBalance = _totalBalance();
```

## CVF-46. INFO

- **Category** Unclear behavior
- **Source** TokenId.sol

**Description** This function silently returns the unmodified value on invalid "i".

**Recommendation** Consider reverting instead.

**Client Comment** *This is only ever called with i values 0-3 in the codebase. Additionally, if you try to clear the 5th leg of an option position it will by definition return itself because there is no 5th leg. Since it is not necessary to check this and revert, we will not be implementing this suggestion.*

```
670 return self;
```

## CVF-47. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This could be simplified as: unchecked{ z = x + y; } require (z >= x); require (uint128 (z) >= uint128 (x));

```
173 unchecked {
        uint128 leftSum = x.leftSlot() + y.leftSlot();
        uint128 rightSum = x.rightSlot() + y.rightSlot();

        if ((leftSum < x.leftSlot()) || (rightSum < x.rightSlot()))
            revert Errors.UnderOverFlow();

180     return z.toRightSlot(rightSum).toLeftSlot(leftSum);
    }
```

## CVF-53. FIXED

- **Category** Flaw
- **Source** PeripheryPayments.sol

**Description** The returned value is ignored.

**Recommendation** Consider explicitly checking that the returned value is true or using the "TransferHelper" library.

**Client Comment** *We no longer use this library.*

```
25 IWETH9(WETH9).transfer(recipient, value);
```

## CVF-54. INFO

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Underflow is possible when converting to "uint256".

**Recommendation** Consider using safe conversion.

**Client Comment** *We now convert to a uint24 and add it to the currentTickPriceFee through a helper function. While underflow is possible, it's acceptable and yields the expected behavior because the variable is only converted to a uint for the purposes of adding the variable to the currentTickPrice, and it is always converted back to an int before any actual use.*

```
395     uint256(int256(currentTick)).addSqrtPrice(sqrtPriceX96).
            ↪ addSwapFee(s_tickSpacing), // 2x Uni v3 feeTier
```

```
422         uint256(int256(currentTick)).addSqrtPrice(sqrtPriceX96),
```

```
439 uint256 currentTickPriceFee = uint256(int256(currentTick)).
        ↪ addSwapFee(s_tickSpacing); // 2x Uni v3 feeTier
```

```
453 uint256 currentTickPriceFee = uint256(int256(currentTick));
```

```
480 uint256 currentTickPriceFee = uint256(int256(currentTick))
```

## CVF-55. FIXED

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Overflow is possible here.

**Recommendation** Consider calculating in 256 bits.

**Client Comment** *The code was refactored such that this multiplication no longer occurs.*

```
482 .addSwapFee(int8(rollITM) * s_tickSpacing); // 2x Uni v3 feeTier
```

## CVF-56. INFO

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math and checked conversion.

**Client Comment** *Overflow is not possible here. The conversion to uint128 will never overflow because the underlying type stored in the slot is always uint64, and the left shift by 64 is a deliberate operation used to pack the two uint64 utilization values into a single uint128.*

```
681 uint128(rateAndUtilization0.leftSlot()) +
    uint128(rateAndUtilization1.leftSlot() << 64);
```

## CVF-57. INFO

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Underflow is possible here.

**Recommendation** Consider using checked math.

**Client Comment** *Underflow is not possible here because the function is only ever called with an offset of 0 or 1, and if called with 1 the positionIdList is always at least 1 element long.*

```
835 pLength = positionIdList.length - offset;
```

## CVF-58. INFO

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math.

**Client Comment** *Overflow is not possible here using the maximum possible values. The maximum value for the numerator is 2**128(max slot size) * 2 ** 160(sqrtprice) which can end up being a maximum of roughly 2**288. However, because we are using mulDiv which calculates in 512 bits, this is fine because we then divide that oversized numerator by the denominator 2**96 which results in an overall max possible value of 2**192, well below the maximum uint256 value of roughly 2**256.*

907
```
) + FullMath.mulDiv(tokenData0.rightSlot(), sqrtPriceX96,
   ↪ FixedPoint96.Q96));
```

914
```
) + FullMath.mulDiv(tokenData0.leftSlot(), sqrtPriceX96,
   ↪ FixedPoint96.Q96));
```

## CVF-59. INFO

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Underflow is possible when converting to “uint256”.

**Recommendation** Consider passing the current tick as a signed integer.

**Client Comment** *We now convert to a uint24 and add it to the currentTickPriceFee through a helper function. While underflow is possible, it’s acceptable and yields the expected behavior because the variable is only converted to a uint for the purposes of adding the variable to the currentTickPrice, and it is always converted back to an int before any actual use.*

935
```
_burnOptions(tokenId, _owner, uint256(int256(currentTick)).
   ↪ addSwapFee(s_tickSpacing)); // 2x Uni v3 feeTier
```

## CVF-60. FIXED

- **Category** Documentation
- **Source** PanopticPool.sol

**Description** This comment looks like an unresolved TODO.

**Recommendation** Consider resolving it.

**Client Comment** *The comment was resolved; no action was needed.*

1395
```
////// DONT WE NEED TO CHECK THAT THE INCOMING LIST == WHAT THE USER
   ↪  ACTUALLY HAS...????
```

## CVF-61. FIXED

- **Category** Overflow/Underflow
- **Source** PanopticPool.sol

**Description** Overflow is possible here.

**Recommendation** Consider using safe conversion.

**Client Comment** *This logic is no longer present. The withdrawal limits by block number were entirely removed.*

1715
```
uint128 newRightSlot = uint128(
```

1717
```
) + (uint128(newBlockNumber) << 32);
```

## CVF-62. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The ”decimals”, ”name”, and ”symbol” properties are optimal in ERC20.

**Recommendation** Consider not assuming that they present in all tokens.

**Client Comment** *We now use try/catch statements to handle tokens that don't implement metadata as expected or at all.*

```
175 s_decimals = IERC20Metadata(underlyingAddress).decimals();
    s_myName = string.concat(PREFIX, IERC20Metadata(underlyingAddress).
       ↪ name());
    s_mySymbol = string.concat(PREFIX, IERC20Metadata(underlyingAddress)
       ↪ .symbol());
```

## CVF-63. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math.

**Client Comment** *It is assumed that the amount of any given token contained within the Panoptic Protocol satisfies the invariant totalAmount < 2**127-1. It is highly unlikely that any mainnet pool will ever fail to satisfy it. With this constraint in mind, overflow is not possible.*

```
294 _newLocked = s_lockedAMM.rightSlot() + amount;
    _newAMM = s_lockedAMM.leftSlot() + _inAMM;
```

## CVF-64. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible here.

**Recommendation** Consider using checked math or performing calculations in 256 bits.

**Client Comment** *We now calculate it in 256 bits.*

```
479 if (currentTick < (strike - range)) {
```

```
488     currNumRangesFromStrike = (2 * (strike - range - currentTick)) /
          ↪  range; // = (strike - range - currentTick) / (range / 2);
          ↪  the "range/2" are the "half ranges"
```

```
498     currNumRangesFromStrike = (2 * (currentTick - strike - range)) /
          ↪  range;
```

```
515 (s_EXERCISE_COST >> uint256(int256(minNumRangesFromStrike))) -
s_COMMISSION_FEE_MAX; // exponential decay of fee based on number of
    ↪  half ranges away from the price
```

## CVF-65. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider calculating in 256-bits.

**Client Comment** *We now calculate it in 256 bits.*

```
521 .toRightSlot((longAmounts.rightSlot() * fee) / DECIMALS_128)
.toLeftSlot((longAmounts.leftSlot() * fee) / DECIMALS_128);
```

## CVF-66. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible here.

**Recommendation** Consider using checked math and safe conversions.

**Client Comment** *It is assumed that the amount of any given token contained within the Panoptic Protocol satisfies the invariant totalAmount < 2**127-1. It is highly unlikely that any mainnet pool will ever fail to satisfy it. With this constraint, overflow is not possible.*

```
538 int256(IERC20(s_underlyingToken).balanceOf(address(s_panopticPool)))
      ↪  - // Total balance in the Panoptic pool
    int256(_lockedFunds()) + // subtract the fees that are reserved for
      ↪ paying fees to options sellers
540 int256(_inAMM()); // add the funds that were moved to the AMM/Uni v3
      ↪  pool
```

## CVF-67. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Underflow is possible when converting to ”uint256”.

**Recommendation** Consider using safe conversion.

**Client Comment** *We have changed the type of inAMM to uint256 to clarify that it cannot be negative.*

```
562 FullMath.mulDiv(uint256(int256(_inAMM())), DECIMALS, _totalBalance()
      ↪ )
```

## CVF-68. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible when converting types.

**Recommendation** Consider using safe conversions.

**Client Comment** *The results of these expressions will never be negative. util < target_util is checked earlier in the function, max sell will always be more than min sell, and saturated will always be more than target. They are also far too small to overflow an int256. There is no potential for overflow here.*

```
611 int256(
613         uint256(int256(max_sell_ratio - min_sell_ratio)),
            uint256(int256(utilization - target_pool_utilization)),
            uint256(int256(saturated_pool_utilization -
               ↪ target_pool_utilization))
```

## CVF-69. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible when converting types.

**Recommendation** Consider using safe conversions.

**Client Comment** *The results of these expressions will never be negative. buy_collat is always positive, and saturated pool utilization is always more than target and the util is checked to be below target earlier. They are also far too small to overflow the type by size. There is no potential for overflow here.*

```
666 int256(
668         uint256(int256(buy_collateral_ratio)),
            uint256(int256(saturated_pool_utilization - utilization)),
670         uint256(int256(saturated_pool_utilization -
               ↪ target_pool_utilization))
```

## CVF-70. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible when converting types.

**Recommendation** Consider using safe conversions.

**Client Comment** *The results of these expressions will never be negative (which would represent an invalid state that is not possible to reach). They are also far too small to overflow the type by size. There is no potential for overflow here.*

715
```
int256(
```

717
```
        uint256(int256(commission_fee_max - commission_fee_min)),
        uint256(int256(utilization - commission_start_utilization)),
        uint256(int256(target_pool_utilization -
            ↪ commission_start_utilization))
```

## CVF-71. FIXED

- **Category** Flaw
- **Source** CollateralTracker.sol

**Description** This check wouldn't work for tokens with zero decimals. Actually, the "decimals" property in ERC-20 is used by UI to render token amounts in human-readable way. Using this property in smart contracts is discouraged.

**Recommendation** Consider just setting the minimum amount of shares in circulation for a non-empty pool to ensure precision. Also note, that a similar attack could be performed with a non-empty pool by removing almost all the liquidity just before a deposit transaction of another user.

**Client Comment** *This check was removed, and we now solicit and effectively burn an initial deposit on creation, ensuring that an attack like this would not be feasible, requiring many multiples of the target in capital for any prospective frontrunner.*

745
```
if ((_totalBalance() == 0) && (_assets < 10**(s_decimals / 2)))
```

## CVF-72. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** Consider setting a minimum amount of shares for a pool and returning ”max (assets, minShares)” here to ensure sufficient precision.

**Client Comment** *We now have a mechanism where an initial deposit is solicited by the factory from the deployer, making a large imbalance between assets and shares leading to precision issues prohibitively costly to create.*

```
830 ? assets
```

## CVF-73. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This would only work in case the delegator has approved tokens to this contract which is weird.

**Recommendation** Consider using ”_transfer” instead.

**Client Comment** *We now uses the internal _transferFrom function of the underlying ERC20.*

```
898 transferFrom(delegator, delegatee, shares);
```

```
941         transferFrom(delegatee, delegator, delegateeBalance);
```

```
947         transferFrom(delegatee, delegator, uint256(requestedAmount))
              ↪ ;
```

## CVF-74. FIXED

- **Category** Flaw
- **Source** CollateralTracker.sol

**Description** The returned value is ignored.

**Recommendation** Consider explicitly requiring the returned value to be true.

**Client Comment** *We no longer use the public transfer function in CollateralTracker, therefore, this issue is no longer valid.*

```
898 transferFrom(delegator, delegatee, shares);
```

```
941         transferFrom(delegatee, delegator, delegateeBalance);
```

```
947         transferFrom(delegatee, delegator, uint256(requestedAmount))
              ↪ ;
```

## CVF-75. INFO

- **Category** Unclear behavior
- **Source** CollateralTracker.sol

**Description** It is unclear why excess collateral cannot be transferred.

**Client Comment** *Excess collateral cannot be transferred normally because that would require a costly calculation on every transfer. We implemented a special redeem function that checks collateral before allowing a withdrawal to accomplish this, so if PLPs must move collateral with open positions, they can simply withdraw through the function and move as they wish.*

```
964 // if they do: we don't want them sending panoptic pool shares to
      ↪ others
    // since that's like reducing collateral
```

## CVF-76. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Under-/overflow is possible here .

**Recommendation** Consider using checked math.

**Client Comment** *None of these values used here will ever be large enough in magnitude to cause an over or underflow.*

1031
```
_addLockedAMM(collectedAmount - premium, shortAmount - longAmount);
```

1034
```
_addLockedAMM(collectedAmount, shortAmount - longAmount);
```

1055
```
tokenToPay = swappedAmount - (shortAmount - longAmount) +
    ↪ commissionAmount;
```

1063
```
tokenToPay -= premium;
```

## CVF-77. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

**Client Comment** *None of these values used here will ever be large enough in magnitude to cause an over or underflow.*

1111
```
int128 _absSwappedAmount = swappedAmount < 0 ? -swappedAmount :
    ↪ swappedAmount;
```

1113
```
    (shortAmount - longAmount + swappedAmount) -
    (_absSwappedAmount * swapRate) /
    DECIMALS_128 -
    currentPositionPremium;
```

1119
```
tokenToPay = -currentPositionPremium;
```

## CVF-78. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

**Client Comment** *We now calculate this in 256 bits.*

```
1181 ((shortAmount + longAmount) * cRate + swappedAmount * swapRate) /
     DECIMALS_128;
```

## CVF-79. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible here when converting types.

**Recommendation** Consider using safe conversion.

**Client Comment** *The maximum pool utilization is 10_000, so this conversion cannot overflow.*

```
1185 uint64 utilization = uint64(uint128(_poolUtilization()));
```

## CVF-80. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible when converting to ”int64”.

**Recommendation** Consider converting line this: int128 (uint128 (utilization))

**Client Comment** *The maximum pool utilization is 10_000, so this conversion cannot overflow.*

```
1205 int128(int64(utilization))
```

## CVF-81. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

**Client Comment** *At this point premiumAllPositions is always negative, so when we flip the sign and then cast to uint it cannot cause an overflow.*

```
1249 tokenRequired += uint128(-premiumAllPositions);
```

## CVF-82. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

**Client Comment** *The total required collateral is assumed to be less than 2**128, so with this constraint in mind no overflow is possible here .*

```
1362 tokenRequired += _tokenRequired;
```

## CVF-83. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Underflow is possible when converting to “uint256”.

**Recommendation** Consider using safe conversion.

**Client Comment** *Because the sell collateral ratio and buy collateral ratio will never be negative, casting these values from an int to a uint cannot cause an overflow.*

```
1390 uint256 sellCollateral = uint256(
```

```
1399 uint256 buyCollateral = uint256(
```

## CVF-84. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible when converting to “int64”.

**Recommendation** Consider using safe conversion.

**Client Comment** *The maximum pool utilization is 10_000, so this conversion cannot overflow.*

1391
```
int256(_sellCollateralRatio(int128(int64(utilization))))
```

1400
```
int256(_buyCollateralRatio(int128(int64(utilization))))
```

## CVF-85. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Underflow is possible when converting to “uint128”.

**Recommendation** Consider using safe conversion.

**Client Comment** *Short/Long amount is always positive so this will never overflow.*

1437
```
_getRequiredCollateralAtUtilization(uint128(shortAmount), 0,
    ↪ utilization) + // short options
_getRequiredCollateralAtUtilization(uint128(longAmount), 1,
    ↪ utilization); // long options
```

## CVF-86. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

**Client Comment** *Short/Long amount is always positive so this will never overflow.*

```
1437 _getRequiredCollateralAtUtilization(uint128(shortAmount), 0,
        ↪ utilization) + // short options
     _getRequiredCollateralAtUtilization(uint128(longAmount), 1,
        ↪ utilization); // long options
```

## CVF-87. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

**Client Comment** *The required collateral will never be more than 2**128-1, so this cannot overflow.*

```
1464 requiredCollateralAsTokens += _tmpCollateralTokensRequired.leftSlot
        ↪ ();
```

## CVF-88. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Client Comment** *The maximum pool utilization is 10_000, so this conversion cannot overflow.*

```
1540 ? uint64(_poolUtilization)
```

## CVF-89. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Underflow is possible here.

**Recommendation** Consider using safe conversion.

**Client Comment** *getRequiredCollateralAtUtilization returns an unsigned integer, so underflow cannot occur. Overflow would not occur because the value of the required collateral is assumed to be less than the maximum int value.*

```
1554 int128(_getRequiredCollateralAtUtilization(amountMoved, 1,
        ↪ utilization))
```

## CVF-90. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible when converting to “int64”.

**Recommendation** Consider using safe conversion.

**Client Comment** *The maximum pool utilization is 10_000, so this conversion cannot overflow.*

```
1563 int128 sellCollateral = _sellCollateralRatio(int128(int64(
        ↪ utilization)));
```

## CVF-91. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible when converting to “int128”.

**Recommendation** Consider using safe conversion.

**Client Comment** *amountMoved is assumed to never exceed the maximum int128 value, so with that constraint in mind this conversion cannot overflow.*

```
1565 required = required.toRightSlot((sellCollateral * int128(amountMoved
        ↪ )) / DECIMALS_128);
```

## CVF-92. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Underflow is possible here.

**Recommendation** Consider using safe conversion.

**Client Comment** *It is not possible for this value to underflow because MulDiv always returns a positive value.*

1615
```
int128(
    int256(
```

1633
```
int128(int256(FullMath.mulDiv(amountMoved, c3, DECIMALS)))
```

## CVF-93. INFO

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Overflow is possible here.

**Recommendation** Consider using safe conversion.

**Client Comment** *amountMoved is assumed to never exceed the maximum int128 value, so in this case it will never overflow.*

1683
```
? required.toLeftSlot(-int128(amountMoved))
: required.toLeftSlot(int128(amountMoved));
```

## CVF-94. FIXED

- **Category** Overflow/Underflow
- **Source** CollateralTracker.sol

**Description** Over-/underflow is possible here.

**Recommendation** Consider using checked math or calculating in 256 bits.

1731
```
((requiredCurrent - requiredBase) * (DECIMALS_128 - requiredBase /
   ↪ 2)) /
(DECIMALS_128 - requiredBase / 2) +
```

## CVF-95. INFO

- **Category** Overflow/Underflow
- **Source** SemiFungiblePositionManager.sol

**Description** Overflow is possible here.

**Recommendation** Consider using safe conversion.

**Client Comment** *The balances are assumed to never exceed the maximum uint128 value, so with that constraint in mind this conversion cannot overflow.*

```
247 uint128 balance = uint128(balanceOf(_msgSender(), tokenId));
```

```
290 uint128 balance = uint128(balanceOf(_msgSender(), oldTokenId));
```

## CVF-96. INFO

- **Category** Overflow/Underflow
- **Source** SemiFungiblePositionManager.sol

**Description** Overflow is possible here when converting to “int128”.

**Client Comment** *The amounts are assumed to never exceed the maximum int128 value, so with that constraint in mind this conversion cannot overflow.*

```
689 movedAmounts = movedAmounts.toRightSlot(-int128(amount0.toUint128())
    ↪ ).toLeftSlot(
690     -int128(amount1.toUint128())
);
```

## CVF-97. INFO

- **Category** Overflow/Underflow
- **Source** SemiFungiblePositionManager.sol

**Description** Underflow is possible here.

**Recommendation** Consider using checked math.

**Client Comment** *As can be seen in the conditional directly above this, these conversions are only run if the value being cast is more than 0, i.e when underflow is impossible.*

```
756 ? receivedAmount0 - uint128(moved.rightSlot())
```

```
759 ? receivedAmount1 - uint128(moved.leftSlot())
```

## CVF-98. INFO

- **Category** Overflow/Underflow
- **Source** FeesCalc.sol

**Description** Over-/underflow is possible when converting types.

**Recommendation** Consider using safe conversion.

**Client Comment** *None of these values will be large enough/negative such that they would exceed their containers or those they are being cast to, so there is no risk for overflow.*

```
214 int128(
        int256(
```

```
225 int128(
        int256(
```

```
365     uint128(uint256(op[index][1])),
```

```
370 fees0 -= int128(
        int256(
```

```
373             uint128(feesToken.rightSlot() - op[index][0].rightSlot()
                    ↪ ),
```

```
379 fees1 -= int128(
380     int256(
```

```
382             uint128(feesToken.leftSlot() - op[index][0].leftSlot()),
```

```
390 fees0 += int128(feesToken.rightSlot() - op[index][0].rightSlot());
    fees1 += int128(feesToken.leftSlot() - op[index][0].leftSlot());
```

## CVF-99. FIXED

- **Category** Overflow/Underflow
- **Source** PanopticMath.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked math or calculating the sum in 256 bits.

**Client Comment** *We now calculate this sum in 256 bits.*

266
```
if (Math.abs(tickUpper + tickLower) < TickMath.MAX_TICK) {
```

## CVF-100. INFO

- **Category** Overflow/Underflow
- **Source** LiquidityChunk.sol

**Description** Overflow is possible here.

**Recommendation** Consider using checked arithmetic or bitwise operations.

**Client Comment** *These functions are only used to build from an empty liquidityChunk, thus, the slots they are adding to will always be 0-initialized, so there is no potential for overflow.*

98
```
return self + uint256(amount);
```

110
```
return self + (uint256(int256(tickLower)) << 232);
```

122
```
return self + ((uint256(int256(tickUpper)) & uint256(BITMASK_INT24))
    ↪  << 208);
```

## CVF-101. INFO

- **Category** Overflow/Underflow
- **Source** TokenId.sol

**Description** Overflow is possible in multiplication.

**Recommendation** Consider adding a range check for the ”legIndex” argument.

**Client Comment** *The only values used for legIndex in the codebase are 0,1,2, or 3. Adding a range check would simply waste gas.*

```
119 return uint256((self >> (80 + legIndex * 4)) % 8);
134 return uint256((self >> (80 + legIndex * 4 + 3)) % 2);
149 return uint256((self >> (96 + legIndex * 40)) % 2);
161 return uint256((self >> (96 + legIndex * 40 + 1)) % 2);
179 return uint256((self >> (96 + legIndex * 40 + 2)) % 4);
191 return int24(int256(self >> (96 + legIndex * 40 + 4)));
204 return int24(int256((self >> (96 + legIndex * 40 + 28)) % 4096));
239 return self + (uint256(_optionRatio % 8) << (80 + legIndex * 4));
257 return self + (uint256(_numeraire % 2) << (80 + legIndex * 4 + 3));
278 return self + (uint256(_isLong % 2) << (96 + legIndex * 40));
294 return self + (uint256(_tokenType % 2) << (96 + legIndex * 40 + 1));
310 return self + (uint256(_riskPartner % 4) << (96 + legIndex * 40 + 2)
    ↪ );
326 return self + uint256((int256(_strike) & BITMASK_INT24) << (96 +
    ↪ legIndex * 40 + 4));
343 return self + (uint256(uint24(_width) % 4096) << (96 + legIndex * 40
    ↪  + 28));
```

## CVF-102. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** This flips the ”isLong” bits even for inactive legs making the whole value inconsistent, as a comment above says that an inactive leg has all bits set to zero.

**Recommendation** Consider either changing the comment above to allow ”isLong” bit for an inactive leg to be non-zero, or correcting this logic to not affect inactive legs.

**Client Comment** *We corrected the logic to not affect inactive legs*

```
362 return self ^ LONG_MASK;
```

## CVF-103. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** This may count inactive legs, as an inactive leg may have the ”isLong” flag set to true.

**Client Comment** *All inactive bits are checked to ensure they are not set now.*

```
374 return self.isLong(0) + self.isLong(1) + self.isLong(2) + self.
    ↪ isLong(3);
```

## CVF-104. INFO

- **Category** Overflow/Underflow
- **Source** TokenId.sol

**Description** Overflow is possible during multiplication.

**Recommendation** Consider using checked math, or performing multiplication in 256 bits.

**Client Comment** *Overflow is not possible because the maximum supported width is 4095 and the maximum supported tick spacing is 200.*

```
395 int24 oneSidedRange = (self.width(legIndex) * tickSpacing) / 2;
```

## CVF-105. FIXED

- **Category** Flaw
- **Source** TokenId.sol

**Recommendation** It should also be ensured that the ”numeraire” bits for the upper legs are zero. Otherwise, the ”countLegs” function would be screwed.

```
459 if (self.optionRatio(j) != 0) revert Errors.InvalidTokenIdParameter
       ↪ (1);
```

## CVF-106. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** This also compares inactive legs, that are not guaranteed to be zeroed in the current implementation.

**Recommendation** Consider either not comparing them or guaranteeing them to be zeroed.

**Client Comment** *Inactive legs are now guaranteed to be zeroed.*

```
592 return ((oldTokenId & ROLL_MASK) == (newTokenId & ROLL_MASK));
```

## CVF-107. INFO

- **Category** Documentation
- **Source** TokenId.sol

**Description** When clearing a leg, this function doesn’t clear the consequent legs, thus it may produce gaps.

**Recommendation** Consider clearly documenting this behavior.

**Client Comment** *This behavior is clearly stated in the NatSpec and its function is also apparent in the one place where it is used. In our opinion, further documentation is not required.*

```
661 function clearLeg(uint256 self, uint256 i) internal pure returns (
       ↪ uint256) {
```

## CVF-108. INFO

- **Category** Overflow/Underflow
- **Source** TickPriceFeeInfo.sol

**Description** Overflow is possible here.

**Recommendation** Consider adding overflow checks.

**Client Comment** *This function is not intended to handle existing bits in the slot it's writing to. Overflow will not occur as long as the slot is 0-initialized.*

```
78 return self + uint256(int256(currentTick) & BITMASK_INT24);
```

```
88 return self + (uint256(sqrtPriceX96) << 24);
```

```
98 return self + ((uint256(int256(_swapFee) & BITMASK_INT24)) << 184);
```

## CVF-111. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** Should be ">" rather than ">=".

**Client Comment** *We changed the operator to ">".*

```
377 if (self >= uint256(type(int256).max)) revert Errors.CastingError();
```

# 9 Minor Issues

## CVF-109. INFO

- **Category** Overflow/Underflow
- **Source** LeftRight.sol

**Description** This may overflow into the left slot.

**Recommendation** Consider either implementing some protection against this or clearly describing this behavior.

**Client Comment** *This function is not intended to handle existing bits in the slot it's writing to, so we clarified this in a comment. We have 'add' and 'sub' functions for this purpose.*

```
56 return self + uint256(right);
```

```
69 return self + uint256(int256(right));
```

```
81 return self + int256(uint256(right));
```

```
94 return self + (int256(right) & RIGHT_HALF_BIT_MASK);
```

## CVF-110. INFO

- **Category** Overflow/Underflow
- **Source** LeftRight.sol

**Description** Overflow is possible here.

**Client Comment** *This function is not intended to handle existing bits in the slot it's writing to, so we clarified this in a comment. We have 'add' and 'sub' functions for this purpose.*

```
132 return self + (uint256(left) << 128);
```

```
144 return self + (int256(int128(left)) << 128);
```

```
156 return self + (int256(left) << 128);
```

## CVF-118. INFO

- **Category** Bad datatype
- **Source** PeripheryPayments.sol

**Recommendation** The type of this argument should be “IERC20”.

**Client Comment** *We no longer use this library.*

```
17 address token,
```

## CVF-119. FIXED

- **Category** Suboptimal
- **Source** PeripheryPayments.sol

**Description** This branch prefers wrapping plain ether into WETH rather than using existing “WETH” balance.

**Recommendation** Consider preferring existing “WETH”.

**Client Comment** *We no longer use this library.*

```
22 if (token == WETH9 && address(this).balance >= value) {
```

## CVF-130. FIXED

- **Category** Suboptimal
- **Source** PanopticFactory.sol

**Description** Specifying a particular compiler version makes it harder to migrate to newer versions.

**Recommendation** Consider specifying as ”^0.8.0”. Also relevant for: FeesCalc.sol, CollateralTracker.sol, PanopticMath.sol, Math.sol, LiquidityChunk.sol, TokenId.sol, TickPriceFeeInfo.sol, LeftRight.sol, Errors.sol, ISemiFungiblePositionManager.sol, IPanopticPool.sol.

**Client Comment** *We opted to loosen the pragma for certain non-core contracts in the interest of composability. For now, the core contracts will remain on a specific version in the interest of security.*

```
2 pragma solidity =0.8.17;
```

## CVF-131. FIXED

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The type of this variable should be “IUniswapV3Factory”.

```
34 address private immutable univ3Factory;
```

## CVF-132. FIXED

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The type of this mapping should be: mapping(IUniswapV3Pool ⇒ IPanopticPool)”.

```
40 mapping(address => address) private s_getPanopticPool;
```

## CVF-133. INFO

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The type of this variable should be “IUniswapV3Pool”.

**Client Comment** *As a call is not made directly with this type, we won’t be implementing this change in the interest of composability.*

```
44 address private immutable POOL_REFERENCE;
```

## CVF-134. INFO

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The type of this variable should be “CollateralTracker” or an interface extracted from it.

**Client Comment** *As a call is not made directly with this type, we won’t be implementing this change in the interest of composability.*

```
46 address private immutable COLLATERAL_REFERENCE;
```

## CVF-135. FIXED

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The argument types should be “ISemiFungiblePositionManager” and “IUniswapV3Factory” respectively.

```
65 constructor(address _SFPM, address _univ3Factory) ERC1155("") {
```

## CVF-136. INFO

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The type of these arguments should be “IERC20”.

**Client Comment** *In the interest of composability we will not be implementing these changes. The documentation makes it very clear what this variable represents.*

```
118 address token0,
    address token1,
```

```
208 address _token0,
    address _token1,
```

```
273 address token0,
    address token1,
```

## CVF-137. FIXED

- **Category** Documentation
- **Source** PanopticFactory.sol

**Description** The semantics of this argument is unclear.

**Recommendation** Consider documenting.

```
214 uint256 minTargetRarity
```

## CVF-138. FIXED

- **Category** Bad naming
- **Source** PanopticFactory.sol

**Description** The semantics of the returned values is unclear.

**Recommendation** Consider giving descriptive names to the returned values and/or explaining in the documentation comment.

```
215 ) external view returns (uint256, uint256) {
```

## CVF-139. FIXED

- **Category** Suboptimal
- **Source** PanopticFactory.sol

**Description** Passing some big number (bigger than 64) as “minTargetRarity” would have the same effect as passing zero.

**Recommendation** Consider removing the comparison with zero to save gas.

```
239 if ((minTargetRarity > 0) && rarity >= minTargetRarity) {
```

## CVF-140. INFO

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The type of this argument should be “IUniswapV3Pool”.

**Client Comment** *We opted not to implement this in the interest of composability. The value is clearly documented.*

```
288 address v3Pool,
```

## CVF-141. INFO

- **Category** Bad datatype
- **Source** PanopticFactory.sol

**Recommendation** The conversion to “address” is redundant as “v3Pool” is already “address”.

**Client Comment** *We opted not to implement this in the interest of composability. The value is clearly documented.*

```
292 return keccak256(abi.encodePacked(address(v3Pool), deployer, nonce))
    ↪ ;
```

## CVF-142. INFO

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Recommendation** These variables should be declared as immutable. This would require some refactoring to makes tham assigned in the constructor.

**Client Comment** *We cannot store values in the constructor because this contract is deployed via a proxy.*

```
45 IUniswapV3Pool private s_univ3pool;
address private s_token0;
address private s_token1;
int24 private s_tickSpacing;
```

```
50 CollateralTracker private s_collateralToken0;
CollateralTracker private s_collateralToken1;
```

## CVF-143. INFO

- **Category** Bad datatype
- **Source** PanopticPool.sol

**Recommendation** The type of these variables should be “IERC20”.

**Client Comment** *We opted not to implement this in the interest of composability. The values are clearly documented.*

```
46 address private s_token0;
address private s_token1;
```

## CVF-144. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** Modifier definitions are intermixed with variable definitions.

**Recommendation** Consider placing modifies after variables.

60
```
modifier onlyFactory() {
```

67
```
modifier onlyFactoryOwner() {
```

## CVF-145. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Recommendation** Conversion to “IPanopticFactory” is redundant as “factory” is already “IPanopticFactory”.

68
```
if (_msgSender() != IPanopticFactory(factory).factoryOwner()) revert
    ↪  Errors.NotOwner();
```

## CVF-146. FIXED

- **Category** Documentation
- **Source** PanopticPool.sol

**Description** The semantics of the first key is unclear.

**Recommendation** Consider documenting.

73
```
mapping(address => mapping(uint256 => mapping(uint256 => mapping(
    ↪ uint256 => int256))))
```

## CVF-147. INFO

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Recommendation** It would be more efficient to merge these mappings into a single mapping whose keys are user addresses and values are structs with three fields encapsulating values of the original mappings.

**Client Comment** *While a fair suggestion, we have opted not to implement this at the time of writing due to time constraints. We will keep this under consideration.*

```
73 mapping(address => mapping(uint256 => mapping(uint256 => mapping(
   ↪ uint256 => int256))))
```

```
81 mapping(address => mapping(uint256 => uint256)) private
   ↪ s_positionBalance;
```

```
89 mapping(address => uint256) private s_positionDetails;
```

## CVF-148. FIXED

- **Category** Bad datatype
- **Source** PanopticPool.sol

**Recommendation** The argument type should be “ISemiFungiblePositionManager”.

```
96 constructor(address _sfpm) {
```

## CVF-149. INFO

- **Category** Readability
- **Source** PanopticPool.sol

**Description** The panoptic factory address is implicitly passed as the message sender. Such implicit arguments make code harder to read.

**Recommendation** Consider passing explicitly.

**Client Comment** *We opted not to do this as taking the sender is somewhat more gas efficient, and the contract is only intended to be deployed by the factory.*

```
98 factory = IPanopticFactory(_msgSender()); // only the panoptic
   ↪ factory creates new Panoptic pools
```

## CVF-150. FIXED

- **Category** Bad datatype
- **Source** PanopticPool.sol

**Recommendation** The argument types should be “IUniswapV3Pool” and “CollateralTracker” respectively.

**Client Comment** *The Uniswap V3 pool is now referenced via interface. However, for better composability with the rest of the system, we have decided against modifying the type of the collateral reference here. The documentation makes it clear what contract address type this parameter is representing.*

```
104 function startPool(address _univ3pool, address _collateralReference)
```

## CVF-151. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** This function always returns true.

**Recommendation** Consider returning nothing.

```
108 returns (bool success)
```

## CVF-152. INFO

- **Category** Bad datatype
- **Source** PanopticPool.sol

**Recommendation** The type of the “token” argument should be “IERC20”.

**Client Comment** *The token argument was removed altogether.*

```
146 function updateParameters(address token, uint256 parameterData)
```

## CVF-153. INFO

- **Category** Readability
- **Source** PanopticPool.sol

**Recommendation** Should be “else if”.

**Client Comment** *There is no need for an else because the code would only ever be reached if the previous condition did not evaluate to true.*

```
233 if (tokenIndex == 1) return s_collateralToken1.getPoolData();
```

## CVF-154. INFO

- **Category** Readability
- **Source** PanopticPool.sol

**Recommendation** Should be “else revert”.

**Client Comment** *There is no need for an else because the code would only ever be reached if the previous condition did not evaluate to true.*

```
234 revert Errors.InvalidToken();
```

## CVF-155. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** Misordered ticks could signal a bug in client code.

**Recommendation** Consider reverting on misordered ticks.

**Client Comment** *We no longer perform this check, so it reverts if the ticks are misordered.*

```
352 // If user-supplied tick limits are different, there's a range we
    ↪ want to stay within:
// Swap tick limits if misordered
(tickLimitLow, tickLimitHigh) = tickLimitLow < tickLimitHigh
    ? (tickLimitLow, tickLimitHigh)
    : (tickLimitHigh, tickLimitLow);
```

## CVF-156. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** These functions always return true.

**Recommendation** Consider returning nothing.

```
407 ) external override returns (bool) {
```

```
434 ) external override returns (bool) {
```

```
447 function burnOptions(uint256 tokenId) external override returns (
    ↪ bool) {
```

```
468 ) external override returns (bool) {
```

## CVF-157. FIXED

- **Category** Procedural
- **Source** PanopticPool.sol

**Recommendation** It is a good practice to put the argument name as a comment next to a boolean literal passed as an argument.

**Client Comment** *This flag is no longer passed.*

```
1136 false
```

## CVF-158. FIXED

- **Category** Documentation
- **Source** PanopticPool.sol

**Recommendation** Uniswap.

```
1280 // Get the current tick from the Uniswpa pool
```

## CVF-159. INFO

- **Category** Bad datatype
- **Source** PanopticPool.sol

**Recommendation** The type of this argument should be “IERC20”.

**Client Comment** *This function was removed.*

```
1325 address token,
```

## CVF-160. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Recommendation** Explicit conversion to “int256” is redundant, as compiler does such conversions automatically.

**Client Comment** *The underlying type was changed to uint256.*

```
1379 adjustedAssets = collateralToken.revoke(_msgSender(), delegatee,
       ↪ int256(requestedAmount));
```

## CVF-161. INFO

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** This check makes it redundant for the “_touchedId” argument to be an array.

**Recommendation** Consider turning this argument into an atomic value.

**Client Comment** *The reason why the singular touchedId is an array is so it composes well with the rest of the system: ‘_administrateAccount‘ expects a list of positions to be touched, so is the only way to pass a single position.*

```
1430 if (_touchedId.length != 1) revert Errors.InputListFail();
```

## CVF-162. INFO

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** Relying on business-level constraints makes code more error-prone.

**Recommendation** Consider using safe conversion.

**Client Comment** *Because this value will never be negative, there is no need to do a safe conversion.*

```
1457      delegatedAmounts = uint256(longAmounts); // should never
              ↪ underflow because longAmounts are always positive
  }
```

## CVF-163. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Recommendation** Consider using safe conversion for “exerciseFees”.

**Client Comment** *The event has been changed to emit a signed int instead.*

```
1462  emit ForcedExercised(_msgSender(), _account, _touchedId[0], uint256(
          ↪ exerciseFees));
```

## CVF-164. INFO

- **Category** Bad datatype
- **Source** PanopticPool.sol

**Recommendation** The argument type should be “IERC20”.

**Client Comment** *We opted not to implement this in the interest of composability. The values are clearly documented.*

```
1607  function _getCollateralToken(address token) private view returns (
          ↪ CollateralTracker) {
```

## CVF-165. INFO

- **Category** Readability
- **Source** PanopticPool.sol

**Recommendation** Should be “else if”.

**Client Comment** *For the sake of readability and organization, we have decided to structure our if statements this way. There is no efficiency to be gained by using an else if /else block.*

```
1610 if (token == s_token1) return s_collateralToken1;
```

## CVF-166. INFO

- **Category** Readability
- **Source** PanopticPool.sol

**Recommendation** Should be “else revert”.

**Client Comment** *For the sake of readability and organization, we have decided to structure our if statements this way. There is no efficiency to be gained by using an else if /else block.*

```
1611 revert Errors.InvalidToken();
```

## CVF-167. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Description** This check seems redundant and just wastes gas.

**Recommendation** Consider removing it and doing proper checked conversions instead.

```
1725 if (_getLastRecordedBlockNumber(user) != newBlockNumber) revert
       ↪ Errors.InvalidUserState();
```

## CVF-168. FIXED

- **Category** Suboptimal
- **Source** PanopticPool.sol

**Recommendation** Final conversions to “uint256” are redundant.

```
1746     return uint256(uint32(_getUserOptionsDetails(user).rightSlot()))
            ↪ ;
```

```
1756 return uint256(uint32(_getUserOptionsDetails(user).rightSlot() >>
        ↪ 32));
```

## CVF-169. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** It would be more efficient to make these variables immutable, but this would require some refactoring: i) moving the initialization logic from the ”startToken” function into the constructor and ii) packing name and symbol into 256-bit words.

**Client Comment** *We cannot store values in the constructor because this contract is deployed via a proxy.*

```
50 string private s_myName;
   string private s_mySymbol;
   uint8 private s_decimals;
```

```
60 address private s_owner; // onlyOwner
```

```
65 address private s_underlyingToken;
```

```
69 address private s_univ3token0;
70 address private s_univ3token1;
```

```
73 bool private s_underlyingIsToken0;
```

```
78 IPanopticPool private s_panopticPool;
```

## CVF-170. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** These two variable seem to have the same value.

**Recommendation** Consider merging into one variable.

```
60 address private s_owner; // onlyOwner
```

```
78 IPanopticPool private s_panopticPool;
```

## CVF-171. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** While this variable is private, its value is exposed in every ”ParametersUpdated” event.

**Recommendation** Consider making this variable public and removing it from the event.

**Client Comment** *This variable was removed along with the event being moved into the PanopticPool, which no longer logs the owner.*

```
60 address private s_owner; // onlyOwner
```

## CVF-172. INFO

- **Category** Bad datatype
- **Source** CollateralTracker.sol

**Recommendation** The type of this variable should be ”IERC20”.

**Client Comment** *We opted not to implement this in the interest of composability. The values are clearly documented.*

```
65 address private s_underlyingToken;
```

## CVF-173. INFO

- **Category** Bad datatype
- **Source** CollateralTracker.sol

**Recommendation** The type of these variables should be ”IERC20”.

**Client Comment** *We opted not to implement this in the interest of composability. The values are clearly documented.*

```
69 address private s_univ3token0;
70 address private s_univ3token1;
```

## CVF-174. FIXED

- **Category** Procedural
- **Source** CollateralTracker.sol

**Description** UPPER_CASE is commonly used for constants.

**Recommendation** Consider using camelCase for variables.

**Client Comment** *We no longer use UPPER_CASE for storage variables*

```
91 int128 private s_COMMISSION_FEE_MIN;
   int128 private s_COMMISSION_FEE_MAX;
   int128 private s_COMMISSION_START_UTILIZATION;
```

```
96 int128 private s_SELL_COLLATERAL_RATIO;
   int128 private s_BUY_COLLATERAL_RATIO;
```

```
100 int128 private s_EXERCISE_COST;
    int256 private s_MAINTENANCE_MARGIN_RATIO;
```

```
104 int128 private s_TARGET_POOL_UTILIZATION;
    int128 private s_SATURATED_POOL_UTILIZATION;
```

## CVF-175. INFO

- **Category** Bad naming
- **Source** CollateralTracker.sol

**Recommendation** Events are usually named via nouns, such as ”Parameters”.

**Client Comment** *This is only true in certain style guides. We have chosen to adopt past-ense events as we believe they are more readable.*

```
112 event ParametersUpdated(address indexed owner, uint256 toParameters)
    ↪ ;
```

## CVF-176. FIXED

- **Category** Procedural
- **Source** CollateralTracker.sol

**Recommendation** It is a good practice to put a comment into an empty block to explain why the block is empty.

**Client Comment** *The empty constructor block was removed altogether.*

```
128 constructor() ERC20("", "") {}
```

## CVF-177. INFO

- **Category** Bad datatype
- **Source** CollateralTracker.sol

**Recommendation** The argument type should be IERC20.

**Client Comment** *We opted not to implement this in the interest of composability. The values are clearly documented.*

```
139 function startToken(address underlyingAddress) external {
```

## CVF-178. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This check is redundant as it is anyway possible to pass a dead underlying address.

**Recommendation** Consider removing this check.

```
145 if (underlyingAddress == address(0)) revert Errors.
    ↪ InvalidInputAddress();
```

## CVF-179. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The message sender address is obtained several times.

**Recommendation** Consider obtaining once and reusing.

**Client Comment** *The code now uses msg.sender directly.*

```
148 IUniswapV3Pool uniswapPool = IPanopticPool(_msgSender()).univ3pool()
    ↪ ;
```

```
161 s_owner = _msgSender(); // becomes onlyOwner
```

```
165 s_panopticPool = IPanopticPool(_msgSender());
```

## CVF-180. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** The default parameter values should be named constants.

**Client Comment** *We cannot store these values as constants because this contract is deployed via a proxy*

```
182 s_MAINTENANCE_MARGIN_RATIO = 11_111; // prevents minting of new
       ↪ options at when collateral < 1.1111*required
```

```
184 s_COMMISSION_FEE_MIN = 20; // minimum commission fee when pool
       ↪ utilization > TARGET_POOL_UTILIZATION
    s_COMMISSION_FEE_MAX = 60; // maximum committion fee when pool
       ↪ utilization < COMMISSION_START_UTILIZATION
    s_COMMISSION_START_UTILIZATION = 1_000; // threshold above which the
       ↪  commission fee starts to decrease
```

```
188 s_SELL_COLLATERAL_RATIO = 2_000; // basal collateral ratio for
       ↪ selling an option (20% of notional)
    s_BUY_COLLATERAL_RATIO = 1_000; // basal collateral ratio for buying
       ↪  an option (10% of notional)
```

```
191 s_TARGET_POOL_UTILIZATION = 5_000; // Target pool utilization where
       ↪ buying+selling is optimal
    s_SATURATED_POOL_UTILIZATION = 9_000; // Pool utilization above
       ↪ which selling is 100% collateral backed
```

```
194 s_EXERCISE_COST = -1_024; // basal cost to force exercise a position
       ↪  that is barely far-the-money (out-of-range).
```

## CVF-181. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This effectively means that the function never returns any value other than true.

**Recommendation** Consider returning nothing.

```
208 * @return true if the update went through without reverts
```

## CVF-182. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The final widening conversions are redundant, as compiler does such conversions automatically.

**Recommendation** Consider removing the final conversions.

**Client Comment** *This code has been refactored to take a struct, so the conversions are no longer needed.*

```
228     ((s_MAINTENANCE_MARGIN_RATIO = int256(int16(uint16(parameterData
           ↪ )))) <= 0) ||
        ((s_COMMISSION_FEE_MIN = int128(int16(uint16(parameterData >>
           ↪ 16)))) <= 0) ||
230     ((s_COMMISSION_FEE_MAX = int128(int16(uint16(parameterData >>
           ↪ 32)))) <= 0) ||
        ((s_COMMISSION_START_UTILIZATION = int128(int16(uint16(
           ↪ parameterData >> 48)))) <= 0) ||
        ((s_SELL_COLLATERAL_RATIO = int128(int16(uint16(parameterData >>
           ↪  64)))) <= 0) ||
        ((s_BUY_COLLATERAL_RATIO = int128(int16(uint16(parameterData >>
           ↪ 80)))) <= 0) ||
```

```
235     ((s_TARGET_POOL_UTILIZATION = int128(int16(uint16(parameterData
           ↪ >> 128)))) <= 0) ||
        ((s_SATURATED_POOL_UTILIZATION = int128(int16(uint16(
           ↪ parameterData >> 144)))) <= 0)
```

```
239 s_EXERCISE_COST = int128(int16(uint16(parameterData >> 96)));
```

## CVF-183. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** Here a value that was just written into the storage is read back again.

**Recommendation** Consider using the written value instead.

**Client Comment** *This code is no longer present.*

```
240 if (s_EXERCISE_COST >= 0) revert Errors.InvalidInputParameters();
```

## CVF-184. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The functions "_inAMM" and "_lockedFunds" are called twice: once another time inside the "_totalBalance" function.

**Recommendation** Consider refactoring to call them only once.

**Client Comment** *We've opted not to refactor this in the interest of increasing readability and reducing complexity.*

```
272 currentTotalBalance = _totalBalance();
    insideAMM = _inAMM();
    totalLocked = _lockedFunds();
```

## CVF-185. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** This could be optimized as: if (amount | _inAMM == 0) return;

```
288 if ((amount == 0) && (_inAMM == 0)) return;
```

## CVF-186. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This expression is calculated twice.

**Recommendation** Consider calculating once and reusing.

**Client Comment** *This expression is no longer calculated twice.*

```
361     FullMath.mulDiv(tokenData.rightSlot(), FixedPoint96.Q96,
          ↪ sqrtPriceX96) +
```

```
376 FullMath.mulDiv(tokenData.rightSlot(), FixedPoint96.Q96,
      ↪ sqrtPriceX96),
```

## CVF-187. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** While the “muldiv” function is very efficient in general case, for specific cases more efficient approaches do exist. For example, when numerator or denominator is a power of 2, multiplication or division could be replaced by shift. When denominator is a compile-time constant, the reciprocal of the denominator could be precomputed.

361
```
    FullMath.mulDiv(tokenData.rightSlot(), FixedPoint96.Q96,
       ↪ sqrtPriceX96) +
    FullMath.mulDiv(otherTokenData.rightSlot(), sqrtPriceX96,
       ↪ FixedPoint96.Q96);
```

370
```
    FullMath.mulDiv(tokenData.leftSlot(), FixedPoint96.Q96,
       ↪ sqrtPriceX96) +
    FullMath.mulDiv(otherTokenData.leftSlot(), sqrtPriceX96,
       ↪ FixedPoint96.Q96);
```

376
```
FullMath.mulDiv(tokenData.rightSlot(), FixedPoint96.Q96,
   ↪ sqrtPriceX96),
```

391
```
            FullMath.mulDiv(
                (tokenValue) * (DECIMALS - valueRatio1),
                FixedPoint96.Q96,
                sqrtPriceX96
            )
```

402
```
            FullMath.mulDiv(
                (requiredValue - tokenValue) * (DECIMALS -
                   ↪ valueRatio1),
                FixedPoint96.Q96,
                sqrtPriceX96
            )
```

415
```
            FullMath.mulDiv(
                (tokenValue) * (valueRatio1),
                sqrtPriceX96,
                FixedPoint96.Q96
            )
```

(... 426, 1395, 1404, 1442, 1617, 1633)

## CVF-188. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** These two code fragments are very similar.

**Recommendation** Consider merging using the ternary operator.

```
389 bonus0 = SafeCast.toInt128(
390     int256(
          FullMath.mulDiv(
              (tokenValue) * (DECIMALS - valueRatio1),
              FixedPoint96.Q96,
              sqrtPriceX96
          )
      )
  );
```

```
400 bonus0 = SafeCast.toInt128(
      int256(
          FullMath.mulDiv(
              (requiredValue - tokenValue) * (DECIMALS - valueRatio1),
              FixedPoint96.Q96,
              sqrtPriceX96
          )
      )
  );
```

## CVF-189. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** These two code fragments are very similar.

**Recommendation** Consider merging using the ternary operator.

```
413 bonus1 = SafeCast.toInt128(
        int256(
            FullMath.mulDiv(
                (tokenValue) * (valueRatio1),
                sqrtPriceX96,
                FixedPoint96.Q96
            )
420     )
    );
```

```
424 bonus1 = SafeCast.toInt128(
        int256(
            FullMath.mulDiv(
                (requiredValue - tokenValue) * (valueRatio1),
                sqrtPriceX96,
                FixedPoint96.Q96
430         )
        )
    );
```

## CVF-190. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This loses 1 bit of precision.

**Recommendation** Consider calculating upperRange and lowerRange separately so their sum is exactly width * s_tickSpacing.

**Client Comment** *At the time of this audit, we've decided to drop support for 1bps pools (or any pool where tickSpacing is not defined by swapFee * 2 / 100 ). In the protocol's current state, we make assumptions about the tickSpacing that are broken by 1bps pools. These are composed of a select few stablecoin pairs, and the vast majority of pairs satisfy our assumptions. Support can be reenabled for those pools in the future once changes are made to stop relying on those assumptions. This will not lose 1 bit of precision if our assumption that the tickSpacing of a pool is divisible by 2 is met.*

```
475 int24 range = (width * s_tickSpacing) / 2;
```

## CVF-191. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** This assignment is redundant, as the assigned value is never used.

```
477 int24 currNumRangesFromStrike = minNumRangesFromStrike;
```

## CVF-192. INFO

- **Category** Procedural
- **Source** CollateralTracker.sol

**Recommendation** Brackets around multiplication are redundant.

**Client Comment** *Although these may be redundant, they assist readers in visualizing the order of operations and increase readability of the code. Generally, it's always better to err on the side of using parentheses for clarity rather than relying on an implicit order of operations.*

488
```
        currNumRangesFromStrike = (2 * (strike - range - currentTick
           ↪ )) / range; // = (strike - range - currentTick) / (
           ↪ range / 2); the "range/2" are the "half ranges"
```

498
```
        currNumRangesFromStrike = (2 * (currentTick - strike - range
           ↪ )) / range;
```

521
```
.toRightSlot((longAmounts.rightSlot() * fee) / DECIMALS_128)
.toLeftSlot((longAmounts.leftSlot() * fee) / DECIMALS_128);
```

## CVF-193. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** Here "10" should be a named constant.

**Client Comment** *This code was removed.*

510
```
minNumRangesFromStrike = minNumRangesFromStrike > 10 ? int24(10) :
   ↪ minNumRangesFromStrike;
```

## CVF-194. INFO

- **Category** Procedural
- **Source** CollateralTracker.sol

**Recommendation** Brackets around multiplication are redundant.

**Client Comment** *Although these may be redundant, they assist readers in visualizing the order of operations and increase readability of the code. Generally, it's always better to err on the side of using parentheses for clarity rather than relying on an implicit order of operations*

```
521 .toRightSlot((longAmounts.rightSlot() * fee) / DECIMALS_128)
    .toLeftSlot((longAmounts.leftSlot() * fee) / DECIMALS_128);
```

## CVF-195. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** Double type conversion is redundant. Just do: uint128 (_inAMM())

```
562 FullMath.mulDiv(uint256(int256(_inAMM())), DECIMALS, _totalBalance()
      ↪ )
```

## CVF-196. FIXED

- **Category** Procedural
- **Source** CollateralTracker.sol

**Description** Here _msgSender() is guaranteed to be the owner, and owner is assumed to be the pool, however such assumption may turn wrong in the future.

**Recommendation** Consider using "s_panopticPool" instead of "_msgSender()".

**Client Comment** *The function is no longer gated by the Panoptic Pool, and the corresponding changes were made, so this is no longer an issue.*

```
754 TransferHelper.safeTransferFrom(s_underlyingToken, _user, _msgSender
      ↪ (), _assets);
```

```
810 TransferHelper.safeTransferFrom(s_underlyingToken, _msgSender(),
      ↪ _user, _assets);
```

## CVF-197. FIXED

- **Category** Procedural
- **Source** CollateralTracker.sol

**Recommendation** This commented out code should be removed.

892
```
//if (s_delegation[delegatee][delegator] != 0) revert Errors.
   ↪ DelegationError();
```

895
```
//s_delegation[delegator][delegatee] += shares;
```

## CVF-198. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The expression ”balanceOf(optionOwner)” is calculated twice.

**Recommendation** Consider calculating once and reusing.

1126
```
sharesToBurn = sharesToBurn <= balanceOf(optionOwner)
```

1128
```
    : balanceOf(optionOwner);
```

## CVF-199. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** ”1” here should be a named constant or even enum constant.

**Client Comment** *This is only called once now, so we are going to leave it as a number at this time. We may refactor in the future.*

1198
```
1,
```

## CVF-200. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The expression ”tokenId.tokenType(index)” is calculated twice.

**Recommendation** Consider calculating once and reusing.

```
1286 bool requireToken0 = (tokenId.tokenType(index) == 0) &&
        ↪ underlyingIsToken0;
     bool requireToken1 = (tokenId.tokenType(index) == 1) &&
        ↪ underlyingIsToken1;
```

## CVF-201. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** The expression “positionIdList.length - offset” is calculated on every loop iteration.

**Recommendation** Consider calculating once.

```
1339 for (uint256 i = 0; i < (positionIdList.length - offset); ) {
```

## CVF-202. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** The variable “_poolUtilization” should be assigned only if “positionSize” is not zero.

```
1344 (uint128 positionSize, uint128 _poolUtilization) = (
```

## CVF-203. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** Explicit conversion to “int128” is redundant.

```
1391 int256(_sellCollateralRatio(int128(int64(utilization))))
```

```
1400 int256(_buyCollateralRatio(int128(int64(utilization))))
```

## CVF-204. INFO

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Recommendation** Should be “else revert” for completeness.

**Client Comment** *Since this is a private function, it is only ever called internally. There will never be a case where an input into isLong is not 0 or 1. As there is no efficiency to be gained here, we will not be making this change.*

```
1405 }
```

## CVF-205. FIXED

- **Category** Suboptimal
- **Source** CollateralTracker.sol

**Description** One bit of precision is lost here.

**Recommendation** Consider calculating upperRange and lowerRange separately.

**Client Comment** *At the time of this audit, we’ve decided to drop support for 1bps pools (or any pool where tickSpacing is not defined by swapFee * 2 / 100 ). In the protocol’s current state, we make assumptions about the tickSpacing that are broken by 1bps pools. These are composed of a select few stablecoin pairs, and the vast majority of pairs satisfy our assumptions. Support can be reenabled for those pools in the future once changes are made to stop relying on those assumptions. This will not lose 1 bit of precision if our assumption that the tickSpacing of a pool is divisible by 2 is met.*

```
1559 int24 oneSidedRange = (tokenId.width(index) * s_tickSpacing) / 2;
```

## CVF-206. INFO

- **Category** Procedural
- **Source** CollateralTracker.sol

**Recommendation** Brackets are redundant around multiplication.

**Client Comment** *Although these may be redundant, they assist readers in visualizing the order of operations and increase readability of the code. Generally, it's always better to err on the side of using parentheses for clarity rather than relying on an implicit order of operations*

```
1565 required = required.toRightSlot((sellCollateral * int128(amountMoved
        ↪ )) / DECIMALS_128);
```

## CVF-207. FIXED

- **Category** Bad datatype
- **Source** SemiFungiblePositionManager.sol

**Recommendation** The type of this variable should be "IWETH9".

**Client Comment** *This code was removed.*

```
55 address private immutable _WETH;
```

## CVF-208. FIXED

- **Category** Suboptimal
- **Source** SemiFungiblePositionManager.sol

**Description** These two structures are identical.

**Recommendation** Consider merging into one.

```
60 struct MintCallbackData {
```

```
66 struct SwapCallbackData {
```

## CVF-209. FIXED

- **Category** Bad datatype
- **Source** SemiFungiblePositionManager.sol

**Recommendation** The argument types should be “IUniswapV3Pool” and “IWETH9” respectively.

**Client Comment** *The WETH argument was removed, and the interface is now used on the factory parameter.*

```
92 constructor(address uniswapFactory, address _WETH9)
```

## CVF-210. INFO

- **Category** Bad datatype
- **Source** SemiFungiblePositionManager.sol

**Recommendation** The type of these arguments should be “IERC20”.

**Client Comment** *We opted not to implement this in the interest of composability. The values are clearly documented.*

```
115 address token0,
    address token1,
```

## CVF-211. FIXED

- **Category** Suboptimal
- **Source** SemiFungiblePositionManager.sol

**Recommendation** These “unchecked” blocks are redundant as there are no operation inside that could be checked.

**Client Comment** *The logic of this function has significantly changed since the time of audit, but there are no longer any unchecked blocks without a purpose there.*

```
515     unchecked {
```

```
523 unchecked {
```

```
544 unchecked {
```

## CVF-212. FIXED

- **Category** Suboptimal
- **Source** SemiFungiblePositionManager.sol

**Recommendation** The value 10 should be a named constant.

**Client Comment** *Named as DUST_THRESHOLD.*

```
686 if ((amount0 < 10) && (amount1 < 10)) revert Errors.
       ↪ NotEnoughLiquidity();
```

## CVF-213. FIXED

- **Category** Documentation
- **Source** SemiFungiblePositionManager.sol

**Description** The comment and the code don't match. The code just puts negated amounts into "movedAmounts".

```
688 // increment the amountsOut with minted amounts. amountsOut =
       ↪ notional value of tokens moved
    movedAmounts = movedAmounts.toRightSlot(-int128(amount0.toUint128())
       ↪ ).toLeftSlot(
690     -int128(amount1.toUint128())
    );
```

## CVF-214. FIXED

- **Category** Suboptimal
- **Source** SemiFungiblePositionManager.sol

**Recommendation** This could be simplified as: bool swapMint = moved0 > 0 && amount0 != 0 || !swapMint && moved1 > 0 && amount1 != 0;

**Client Comment** *This logic has been changed sufficiently enough that the simplification given is no longer relevant.*

```
854 bool swapMint;
    if ((moved0 > 0) && (amount0 != 0)) {
        swapMint = true;
    }
    if (!swapMint && ((moved1 > 0) && (amount1 != 0))) {
        swapMint = true;
860 }
```

## CVF-215. INFO

- **Category** Bad datatype
- **Source** SemiFungiblePositionManager.sol

**Recommendation** The type of this argument should be “IUniswapV3Pool”.

**Client Comment** *As there is no call to the pool made in this function, we do not need the interface type here and we will not be implementing this change.*

```
980 address univ3poolAddress,
```

## CVF-216. FIXED

- **Category** Documentation
- **Source** FeesCalc.sol

**Description** The semantics of keys and values in these mappings is unclear.

**Recommendation** Consider documenting.

```
68  mapping(uint256 => mapping(uint256 => mapping(uint256 => int256)))
        ↪ storage userOptions,
    mapping(uint256 => uint256) storage userBalance,
```

```
119 mapping(uint256 => mapping(uint256 => int256)) storage userOptions,
```

```
348 mapping(uint256 => mapping(uint256 => int256)) storage op,
```

## CVF-217. FIXED

- **Category** Procedural
- **Source** FeesCalc.sol

**Description** UPPER_CASE is commonly used for constants.

**Recommendation** Consider using camelCase for arguments.

**Client Comment** *We now reserve UPPER_CASE for constants.*

```
71  bool ALL_PREMIA_FLAG
```

## CVF-218. FIXED

- **Category** Suboptimal
- **Source** FeesCalc.sol

**Description** These variables are not used outside the loop.

**Recommendation** Consider moving their definitions into the loop.

```
74  uint256 tokenId;
    int256 positionPremia;
    uint128 positionSize;
```

## CVF-219. FIXED

- **Category** Suboptimal
- **Source** FeesCalc.sol

**Description** While the ”muldiv” function is very efficient in general case, more efficient approaches exist for specific cases, such as when the denominator is a power of two known at compile time, or when the denominator is a compile-time constant. In the former case, division could be replaced with shift and in the latter case the reciprocal of the denominator could be precomputed.

**Recommendation** Consider using more efficient approaches when applicable.

```
216                 FullMath.mulDiv(
                        ammFeesPerLiqToken0X128,
                        liquidityChunk.liquidity(),
                        FixedPoint128.Q128
220                 )
```

```
227                 FullMath.mulDiv(
                        ammFeesPerLiqToken1X128,
                        liquidityChunk.liquidity(),
230                     FixedPoint128.Q128
                    )
```

```
372                 FullMath.mulDiv(
                        uint128(feesToken.rightSlot() - op[index][0].
                            ↪ rightSlot()),
                        effectiveLiquidityFactor,
                        DECIMALS
                    )
```

```
381                 FullMath.mulDiv(
                        uint128(feesToken.leftSlot() - op[index][0].
                            ↪ leftSlot()),
                        effectiveLiquidityFactor,
                        DECIMALS
                    )
```

```
419 effectiveLiquidityFactor = FullMath.mulDiv(baseLiquidity, DECIMALS,
       ↪ baseLiquidity - amount);
```

## CVF-220. FIXED

- **Category** Suboptimal
- **Source** FeesCalc.sol

**Description** The expression ”liquidityChunk.liquidity()” is calculated twice.

**Recommendation** Consider calculating once and reusing.

```
218 liquidityChunk.liquidity(),
```

```
229 liquidityChunk.liquidity(),
```

## CVF-221. FIXED

- **Category** Suboptimal
- **Source** FeesCalc.sol

**Description** These values are needed in both branches of the conditional statement.

**Recommendation** Consider calculating in once place before the conditional statement.

```
373             uint128(feesToken.rightSlot() - op[index][0].rightSlot()
                ↪ ),
```

```
382             uint128(feesToken.leftSlot() - op[index][0].leftSlot()),
```

```
390 fees0 += int128(feesToken.rightSlot() - op[index][0].rightSlot());
    fees1 += int128(feesToken.leftSlot() - op[index][0].leftSlot());
```

## CVF-222. INFO

- **Category** Bad datatype
- **Source** PanopticMath.sol

**Recommendation** The argument type should be ”IUniswapV3Pool”.

**Client Comment** *We do not need to use an interface type, as we don’t directly call the pool from here.*

```
134 function getPoolId(address univ3pool) external pure returns (uint80)
    ↪  {
```

## CVF-223. FIXED

- **Category** Suboptimal
- **Source** PanopticMath.sol

**Description** This function is not used.

**Recommendation** Consider removing it.

**Client Comment** *This function is now used in PanopticFactory.*

```
202 function numberOfLeadingHexZeros(address addr) external pure returns
    ↪  (uint256) {
```

## CVF-224. FIXED

- **Category** Suboptimal
- **Source** PanopticMath.sol

**Description** Calculating a most significant bit index and then converting to a most significant nibble is suboptimal.

**Recommendation** Consider implementing a separate function to for most significant nibble calculation.

```
203 return (159 - BitMath.mostSignificantBit(uint256(uint160(addr)))) /
    ↪ 4;
```

## CVF-225. FIXED

- **Category** Suboptimal
- **Source** PanopticMath.sol

**Description** While the ”muldiv” function is very efficient in general case, more efficient approaches exist for specific cases, such as when the denominator or numerator are powers of 2 known at compile time.

**Recommendation** Consider implementing ”shldiv” and ”mulshr” functions.

**Client Comment** *We've implemented MulDiv96 for this purpose, and transitioned some unnecessary mulDivs that cannot overflow back to normal arithmetic.*

```
238 if (tokenType == 0) return FullMath.mulDiv(balance, FixedPoint96.Q96
       ↪ , sqrtPriceAtTick);
    if (tokenType == 1) return FullMath.mulDiv(balance, sqrtPriceAtTick,
       ↪  FixedPoint96.Q96);
```

```
276         notional = FullMath.mulDiv(contractSize, strikeX96,
               ↪ FixedPoint96.Q96).toUint128();
```

```
278         notional = FullMath.mulDiv(contractSize, FixedPoint96.Q96,
               ↪ strikeX96).toUint128();
```

## CVF-226. INFO

- **Category** Suboptimal
- **Source** Math.sol

**Recommendation** Special handling of the ”type(int256).min” value wouldn't be necessary if the return type would be ”uint256”.

**Client Comment** *We are not implementing the requested change due to time constraints, but we have removed that check as an overflow panic occurs if attempting to evaluate -type(int256).min.*

```
46 if (x == type(int256).min) revert Errors.CoreMathError();
```

## CVF-227. FIXED

- **Category** Suboptimal
- **Source** Math.sol

**Recommendation** This could be simplified as: if ((downcastedInt = uint128(toDowncast)) != toDowncast) ...

```
58 if (!((downcastedInt = uint128(toDowncast)) == toDowncast)) revert
      ↪ Errors.CastingError();
```

## CVF-228. FIXED

- **Category** Suboptimal
- **Source** Math.sol

**Recommendation** This could be simplified as: if ((result = int128(toCase)) < 0) revert ...;

```
67 if (toCast > uint128(type(int128).max)) revert Errors.CastingError()
      ↪ ;
   return int128(toCast);
```

## CVF-229. INFO

- **Category** Suboptimal
- **Source** LiquidityChunk.sol

**Description** This argument is redundant as it is assumed to always be zero.

**Recommendation** Consider removing it.

**Client Comment** *Self refers to the referenced liquidityChunk which is indeed assumed to be zero. This is done so that we can use the LiquidityChunk library on uint256 as a type.*

```
80 uint256 self,
```

## CVF-230. FIXED

- **Category** Suboptimal
- **Source** LiquidityChunk.sol

**Recommendation** This could be simplified as: return self + uint256(uint24(tickUpper)) « 208;

**Client Comment** *This is incorrect, the simplification should be return self + ((uint256(uint24(tickUpper))) « 208). We have implemented the corrected version,*

```
122 return self + ((uint256(int256(tickUpper)) & uint256(BITMASK_INT24))
      ↪  << 208);
```

## CVF-231. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Recommendation** Multiplication and modulo could be replace by bitwise operators.

**Client Comment** *The Solidity compiler replaces arithmetic operations with their more efficient bitwise counterparts whenever possible during the optimization process.*

```
119 return uint256((self >> (80 + legIndex * 4)) % 8);
```

```
134 return uint256((self >> (80 + legIndex * 4 + 3)) % 2);
```

```
239 return self + (uint256(_optionRatio % 8) << (80 + legIndex * 4));
```

```
257 return self + (uint256(_numeraire % 2) << (80 + legIndex * 4 + 3));
```

## CVF-232. INFO

- **Category** Bad datatype
- **Source** TokenId.sol

**Recommendation** The return type should be ”bool”.

**Client Comment** *Due to composability with the other functions in the codebase it is in our best interests to keep the return data type as a uint256. As the return value is used in math operations inside other functions. Bools cannot have math operations explicitly applied to them.*

```
147 function isLong(uint256 self, uint256 legIndex) internal pure
      ↪ returns (uint256) {
```

## CVF-233. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the pool slot of ”self” is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *This code is referenced within the addUniv3pool function. In all places where this function is referenced, the self argument is passed in as a zero value. Thus, there is no reason to add an empty slot check here.*

```
221 return self + uint256(_poolId);
```

## CVF-234. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the ratio slot to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addOptionRatio. There is no need to check that the option ratio bits being written to are zero, as any leg being passed into this function have already been cleared, or are a new uninitalized leg.*

```
239 return self + (uint256(_optionRatio % 8) << (80 + legIndex * 4));
```

## CVF-235. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the numeraire slot to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addNumeraire. There is no need to check that the numeraire bit being written to is zero, as any leg being passed into this function have already been cleared, or are a new uninitalized leg.*

```
257 return self + (uint256(_numeraire % 2) << (80 + legIndex * 4 + 3));
```

## CVF-236. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the ”isLong” flag to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addIsLong. There is no need to check that the isLong bit being written to is zero, as any leg being passed into this function has already been cleared, or are a new uninitalized leg.*

```
278 return self + (uint256(_isLong % 2) << (96 + legIndex * 40));
```

## CVF-237. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the token type slot to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addTokenType. There is no need to check that the tokenType bit being written to is zero, as any leg being passed into this function has already been cleared, or are a new uninitalized leg.*

```
294 return self + (uint256(_tokenType % 2) << (96 + legIndex * 40 + 1));
```

## CVF-238. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the risk partner slot to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addRiskPartner. There is no need to check that the riskPartner bits being written to are zero, as any leg being passed into this function has already been cleared, or are a new uninitalized leg.*

```
310 return self + (uint256(_riskPartner % 4) << (96 + legIndex * 40 + 2)
    ↪ );
```

## CVF-239. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the strike slot to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addStrike. There is no need to check that the strike bits being written to are zero, as any leg being passed into this function have already been cleared, or are a new uninitalized leg.*

```
326 return self + uint256((int256(_strike) & BITMASK_INT24) << (96 +
    ↪ legIndex * 40 + 4));
```

## CVF-240. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no check to ensure that the width slot to be written is empty.

**Recommendation** Consider adding such a check or clearly explaining that the caller should ensure that.

**Client Comment** *The code referenced here is from the function addWidth. There is no need to check that the width bits being written to are zero, as any leg being passed into this function have already been cleared, or are a new uninitalized leg.*

```
343 return self + (uint256(uint24(_width) % 4096) << (96 + legIndex * 40
    ↪  + 28));
```

## CVF-241. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** The expression ”self.width(legIndex)” is calculated twice.

**Recommendation** Consider calculating once and reusing.

```
395 int24 oneSidedRange = (self.width(legIndex) * tickSpacing) / 2;
```

```
401 (legLowerTick, legUpperTick) = self.width(legIndex) == MAX_LEG_WIDTH
```

## CVF-242. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** The expression ”self.strike(legIndex)” is calculated several times.

**Recommendation** Consider calculating once and reusing.

```
397 (self.strike(legIndex) - oneSidedRange <= TickMath.MIN_TICK) ||
    (self.strike(legIndex) + oneSidedRange >= TickMath.MAX_TICK)
```

```
406 : (self.strike(legIndex) - oneSidedRange, self.strike(legIndex) +
    ↪ oneSidedRange);
```

## CVF-243. INFO

- **Category** Procedural
- **Source** TokenId.sol

**Recommendation** Brackets around the first operation are redundant here.

**Client Comment** *Although these may be redundant, they assist readers in visualizing the order of operations and increase readability of the code. Generally, it's always better to err on the side of using parentheses for clarity rather than relying on an implicit order of operations*

395
```
int24 oneSidedRange = (self.width(legIndex) * tickSpacing) / 2;
```

403
```
        (TickMath.MIN_TICK / tickSpacing) * tickSpacing,
        (TickMath.MAX_TICK / tickSpacing) * tickSpacing
```

## CVF-244. FIXED

- **Category** Readability
- **Source** TokenId.sol

**Recommendation** Should be "else if".

436
```
return 4;
```

## CVF-245. FIXED

- **Category** Suboptimal
- **Source** TokenId.sol

**Recommendation** This logical expression could be simplified.

492
```
!(
    (self.tokenType(riskPartnerIndex) == self.tokenType(i) &&
        (self.isLong(riskPartnerIndex) != self.isLong(i)))
) &&
!((self.tokenType(riskPartnerIndex) != self.tokenType(i)) &&
    (self.isLong(riskPartnerIndex) == self.isLong(i)))
```

## CVF-246. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There is no validity check for "i".

**Recommendation** Consider reverting on an invalid "i" value.

**Client Comment** *We have chosen not to revert here because we believe that the behavior of returning an identical tokenId after trying to clear an out-of-range leg index makes sense. If you request to clear index 5, for example, that index does not exist and the leg is already "clear" so it should return the same tokenId.*

```
661 function clearLeg(uint256 self, uint256 i) internal pure returns (
      ↪ uint256) {
```

## CVF-247. INFO

- **Category** Readability
- **Source** TokenId.sol

**Recommendation** Should be "else if".

**Client Comment** *We've organized it this way for readability purposes. We do not need the extra else since it returns in every if statement, and thus will not move on to the next.*

```
664 if (i == 1)
```

```
666 if (i == 2)
```

```
668 if (i == 3) return self & 0
      ↪ xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0FFFFFFFFFFFFFFFFFFFFFFF;
```

## CVF-248. INFO

- **Category** Suboptimal
- **Source** TokenId.sol

**Description** There are no range checks for these arguments.

**Recommendation** Consider adding appropriate checks.

**Client Comment** *This is in reference to the function rollTokenInfo, which has the src and dst arguments passed into it via the constructRollTokenIdWith function. As there are 4 legs indexed 0-3, there is already a definite range set when passing in src and dst.*

```
683 uint256 src,
    uint256 dst
```

## CVF-249. FIXED

- **Category** Suboptimal
- **Source** TickPriceFeeInfo.sol

**Recommendation** This could be simplified as: self + uint24(currentTick)

```
78 return self + uint256(int256(currentTick) & BITMASK_INT24);
```

## CVF-250. FIXED

- **Category** Suboptimal
- **Source** TickPriceFeeInfo.sol

**Recommendation** This could be simplified as: self + (uint256(uint24(_swapFee)) « 184)

```
98 return self + ((uint256(int256(_swapFee) & BITMASK_INT24)) << 184);
```

## CVF-251. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”x.leftSlot()” and ”x.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

```
174 uint128 leftSum = x.leftSlot() + y.leftSlot();
    uint128 rightSum = x.rightSlot() + y.rightSlot();

177 if ((leftSum < x.leftSlot()) || (rightSum < x.rightSlot()))
```

## CVF-252. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”x.leftSlot()” and ”x.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

```
192 uint128 leftSub = x.leftSlot() - y.leftSlot();
    uint128 rightSub = x.rightSlot() - y.rightSlot();

195 if ((leftSub > x.leftSlot()) || (rightSub > x.rightSlot()))
```

## CVF-253. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This could be simplified as: unchecked { uint256 left = uint256 (x.leftSlot ()) * uint256 (y.leftSlot ()); uint128 left128 = uint128 (left); require (left128 == left); z = uint256 (x.rightSlot ()) * uint256 (y.rightSlot ()); require (uint128 (z) == z); return z.toLeftSlot (left128); }

```
209 unchecked {
210     uint128 leftMul = x.leftSlot() * y.leftSlot();
    uint128 rightMul = x.rightSlot() * y.rightSlot();

    if ((x.leftSlot() != 0) && (leftMul / x.leftSlot()) != y.
       ↪ leftSlot())
        revert Errors.UnderOverFlow();
    if ((x.rightSlot() != 0) && (rightMul / x.rightSlot()) != y.
       ↪ rightSlot())
        revert Errors.UnderOverFlow();

    return z.toRightSlot(rightMul).toLeftSlot(leftMul);
}
```

## CVF-254. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”x.leftSlot()”, ”y.leftSlot()”, ”x.rightSlot()”, and ”y.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

```
210 uint128 leftMul = x.leftSlot() * y.leftSlot();
    uint128 rightMul = x.rightSlot() * y.rightSlot();
```

```
213 if ((x.leftSlot() != 0) && (leftMul / x.leftSlot()) != y.leftSlot())
```

```
215 if ((x.rightSlot() != 0) && (rightMul / x.rightSlot()) != y.
       ↪ rightSlot())
```

## CVF-255. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”y.leftSlot()” and ”y.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

```
230 if ((y.leftSlot() == 0) || (y.rightSlot() == 0)) revert Errors.
    ↪ LeftRightInputError();
```

```
233     z.toRightSlot(x.rightSlot() / y.rightSlot()).toLeftSlot(
            x.leftSlot() / y.leftSlot()
```

## CVF-256. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This is not necessary overflow, as ”y” could be zero.

```
247 if (x.leftSlot() == type(uint128).max || x.rightSlot() == type(
    ↪ uint128).max)
     revert Errors.UnderOverFlow();
```

## CVF-257. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”x.leftSlot()”, ”y.leftSlot()”, ”x.rightSlot()”, and ”y.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

247
```
if (x.leftSlot() == type(uint128).max || x.rightSlot() == type(
    ↪ uint128).max)
```

250
```
int128 leftSum = int128(x.leftSlot()) + y.leftSlot();
int128 rightSum = int128(x.rightSlot()) + y.rightSlot();
```

254
```
    ((leftSum < int128(x.leftSlot())) && (y.leftSlot() > 0)) ||
    ((leftSum > int128(x.leftSlot())) && (y.leftSlot() < 0)) ||
    ((rightSum < int128(x.rightSlot())) && (y.rightSlot() > 0)) ||
    ((rightSum > int128(x.rightSlot())) && (y.rightSlot() < 0))
```

## CVF-258. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”x.leftSlot()”, ”y.leftSlot()”, ”x.rightSlot()”, and ”y.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

272
```
int128 leftSum = x.leftSlot() + y.leftSlot();
int128 rightSum = x.rightSlot() + y.rightSlot();
```

276
```
    ((leftSum < x.leftSlot()) && (y.leftSlot() > 0)) ||
    ((rightSum < x.rightSlot()) && (y.rightSlot() > 0)) ||
    ((leftSum > x.leftSlot()) && (y.leftSlot() < 0)) ||
    ((rightSum > x.rightSlot()) && (y.rightSlot() < 0))
```

## CVF-259. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions ”x.leftSlot()”, ”y.leftSlot()”, ”x.rightSlot()”, and ”y.rightSlot()” are calculated twice.

**Recommendation** Consider calculating once and reusing.

```
294 int128 leftSub = x.leftSlot() - y.leftSlot();
    int128 rightSub = x.rightSlot() - y.rightSlot();
```

```
297     ((leftSub > x.leftSlot()) && (y.leftSlot() > 0)) ||
        ((rightSub > x.rightSlot()) && (y.rightSlot() > 0)) ||
        ((leftSub < x.leftSlot()) && (y.leftSlot() < 0)) ||
300     ((rightSub < x.rightSlot()) && (y.rightSlot() < 0))
```

## CVF-260. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This could be simplified as: unchecked { iint256 left = iint256 (x.leftSlot ()) * iint256 (y.leftSlot ()); iint128 left128 = iint128 (left); require (left128 == left); z = iint256 (x.rightSlot ()) * iint256 (y.rightSlot ()); require (iint128 (z) == z); return z.toLeftSlot (left128); }

```
314 unchecked {
        int128 leftMul = x.leftSlot() * y.leftSlot();
        int128 rightMul = x.rightSlot() * y.rightSlot();
        if (
            ((x.leftSlot() != 0) && (leftMul / x.leftSlot()) != y.
               ↪ leftSlot()) ||
            ((x.rightSlot() != 0) && (rightMul / x.rightSlot()) != y.
               ↪ rightSlot())
320     ) revert Errors.UnderOverFlow();

        return z.toRightSlot(rightMul).toLeftSlot(leftMul);
    }
```

## CVF-261. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions "x.leftSlot()", "y.leftSlot()", "x.rightSlot()", and "y.rightSlot()" are calculated twice.

**Recommendation** Consider calculating once and reusing.

315
```
int128 leftMul = x.leftSlot() * y.leftSlot();
int128 rightMul = x.rightSlot() * y.rightSlot();
```

318
```
    ((x.leftSlot() != 0) && (leftMul / x.leftSlot()) != y.leftSlot()
       ↪ ) ||
    ((x.rightSlot() != 0) && (rightMul / x.rightSlot()) != y.
       ↪ rightSlot())
```

## CVF-262. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Description** The expressions "x.leftSlot()", "y.leftSlot()", "x.rightSlot()", and "y.rightSlot()" are calculated twice.

**Recommendation** Consider calculating once and reusing.

334
```
if ((y.leftSlot() == 0) || (y.rightSlot() == 0)) revert Errors.
   ↪ DivisionByZero();
```

336
```
    (x.leftSlot() == type(int128).min && y.leftSlot() == -1) ||
    (x.rightSlot() == type(int128).min && y.rightSlot() == -1)
```

341
```
    z.toLeftSlot(x.leftSlot() / y.leftSlot()).toRightSlot(
        x.rightSlot() / y.rightSlot()
```

## CVF-263. FIXED

- **Category** Suboptimal
- **Source** LeftRight.sol

**Recommendation** This logic is overcomplicated. Just do a 256-bit division of 128-bit integers, and then check, whether the result fits into 128 bits using: int128(x) == x

335
```
if (
    (x.leftSlot() == type(int128).min && y.leftSlot() == -1) ||
    (x.rightSlot() == type(int128).min && y.rightSlot() == -1)
) revert Errors.UnderOverFlow();
```

## CVF-264. INFO

- **Category** Suboptimal
- **Source** Errors.sol

**Recommendation** These errors could be made more useful by adding some parameters to them.

**Client Comment** *While a fair suggestion, we have opted not to refactor our errors at this time. We will keep this in mind as we move forward .*

```
12  error AccountNotSolvent();
18  error AlreadyOwner();
33  error CollateralTokenAlreadyInitialized();
46  error DuplicatedItems();
62  error FirstDepositTooSmall();
89  error InvalidInputAddress();
92  error InvalidInputParameters();
96  error InvalidPanopticPoolState();
99  error InvalidToken();
102 error InvalidTokenAddress();
109 error InvalidUserState();
112 error InvalidValue();
```

ABDK
Consulting

## About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

## Contact


**Email**

dmitry@abdkconsulting.com

**Website**

abdk.consulting

**Twitter**

twitter.com/ABDKconsulting

**LinkedIn**

linkedin.com/company/abdk-consulting