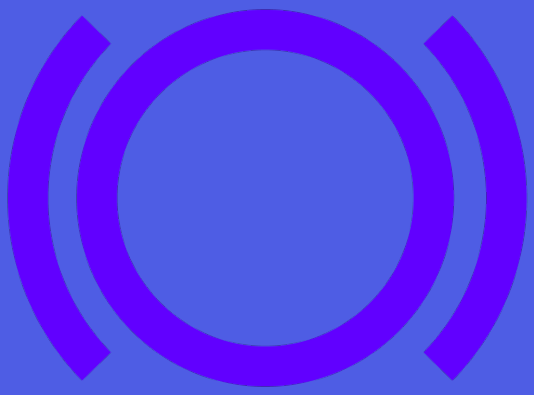
OpenZeppelin | security

Panoptic Audit



December 27, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-03-13 To 2023-04-14 From 2023-05-01 To 2023-05-26 From 2023-06-05 To 2023-07-03
Languages	Solidity

Total Issues	32 (19 resolved)
Critical Severity Issues	4 (4 resolved)
High Severity Issues	3 (3 resolved)
Medium Severity Issues	7 (3 resolved)
Low Severity Issues	13 (6 resolved)
Notes & Additional Information	5 (3 resolved)

Scope

The Panoptic team engaged with us in the past 3 months in its on-going effort to secure its protocol through two phases of audits. This report consolidates the findings with respect to the latest commit in the second phase. We are pleased with the team's effort and emphasis on shipping an innovative, efficient and secure protocol. The Panoptic team has provided us with comprehensive documentation prior to the audit and engaged in regular insightful discussions with the auditors throughout.

This report includes issues from our audit on the `panoptic-labs/Panoptic` repository at the 2d440a9 commit.

Update: The Panoptic team have migrated from a private to a public repository during the audit. As such, some fixes to our findings were resolved in this private repository and are not available. In these cases, we include an update statement without links to commits or PR's.

In scope were the following contracts:

```
contracts
├── CollateralTracker.sol
├── SemiFungiblePositionManager.sol
├── PanopticFactory.sol
├── PanopticPool.sol
├── external
│   ├── uniswapv3_core
│   │   └── contracts
│   │       ├── libraries
│   │       │   ├── BitMath.sol
│   │       │   ├── FullMath.sol
│   │       │   ├── SafeCast.sol
│   │       │   ├── SqrtPriceMath.sol
│   │       │   ├── TickMath.sol
│   │       │   └── UnsafeMath.sol
│   └── uniswapv3_periphery
│       └── contracts
│           └── libraries
│               ├── CallbackValidation.sol
│               ├── LiquidityAmounts.sol
│               ├── PoolAddress.sol
│               ├── PositionKey.sol
│               └── TransferHelper.sol
├── libraries
│   ├── FeesCalc.sol
│   ├── LeftRight.sol
│   ├── LiquidityChunk.sol
```

```
│   ├── Math.sol
│   ├── PanopticMath.sol
│   ├── TickPriceFeeInfo.sol
│   └── TokenId.sol
├── multicall
│   └── Multicall.sol
├── periphery
│   ├── PanopticHelper.sol
│   └── PanopticMigrator.sol
└── tokens
    ├── ERC1155Minimal.sol
    ├── ERC20Minimal.sol
    └── interfaces
        └── IERC20Partial.sol
```

System Overview

Panoptic is a perpetual, oracle-free options protocol for Ethereum. The protocol allows the creation of put and call options in a permissionless manner by largely building on top of the liquidity provisioning and swapping mechanism provided by concentrated liquidity AMMs, currently on Uniswap V3. In the following sections, we will highlight some main aspects in more detail.

Architecture

The system's architecture is split into four main components:

- Semi-fungible Position Manager (SFPM): The engine of the Panoptic protocol, the SFPM is a Uniswap V3 position manager that wraps Uniswap V3 positions behind an ERC-1155 token. The minting and burning of all options are passed to the SFPM to handle liquidity provision and swapping to mint/burn those positions in Uniswap V3.
- Panoptic Pool: The main entry point to the options trading system to buy or sell an option position. Manages positions, collateral, premia streaming, liquidation, and forced exercises.
- Panoptic Factory: Creates and registers Panoptic Pools, similar to the Uniswap V3 factory pool creation pattern. Anyone can create a pool, and only one pool may exist for each corresponding Uniswap V3 pool.
- Collateral Tracker: Collateral tracking and margin accounting used with Panoptic Pools. Each pool has two tokens, each with its own instance of a Collateral Tracker contract. It implements the ERC-4626 standard allowing the minting and burning of shares for underlying assets.

Selling and Buying Panoptic Options

The Panoptic protocol allows the minting of both short and long options of up to 4 legs of tick ranges. Short options create a new liquidity position in Uniswap V3, while a long position removes liquidity. There must be an existing short position with enough available liquidity to

mint a long position at a particular tick range. To mint options positions in a Panoptic Pool, a user must first deposit some collateral to either of the two Collateral Trackers that the pool maintains. This deposit will be in the form of ERC-20 tokens for which the user will receive ERC-4626 shares in return.

Option sellers can always mint positions (with up to x5 leverage) creating available liquidity in particular tick ranges for option buyers. When sellers exercise their positions, they will receive a corresponding premium payout and settle any profit or loss in collateral shares. If the sellers' positions are subsequently bought by buyers, they will not be able to close their positions when the corresponding tick range liquidity is in use. In this case, the seller can force exercise out-of-range long positions to free up liquidity to exit.

Buyers can mint long positions (with up to x20 leverage), provided there is enough available liquidity from option sellers. When buyers exercise their positions, they pay a premium in collateral shares, which are locked and only accessible by option sellers. In order to exercise a long position, collateral tokens must be added back to the Uniswap V3 pool from the Panoptic Pool. With enough available collateral assets, buyers can exercise their positions and get profit/loss via minting or burning their collateral shares.

Liquidity Provisioning in Panoptic Pools

Liquidity providers (LPs) to a Panoptic Pool can deposit collateral assets into their respective collateral trackers. These assets are utilized as leverage by options traders. In return, LPs earn commission fees (10bps) each time an option is minted. A mevTax is required from each deposit to discourage opportune MEV activities. Thus LPs can benefit from the mevTax earnings from each deposit. LPs can redeem their shares for assets if they do not have any open positions. The share price in the collateral tracker may fluctuate through normal operations in the protocol. When options positions are minted or burned, depending on the strike price, new collateral shares can be minted or burned without a respective collateral deposit/withdrawal, thereby changing the collateral share price. Also, during liquidation, a bonus is calculated to pay the liquidator as an incentive for liquidating the distressed account. If the distressed account's collateral is not able to cover the bonus amount, new shares will be minted to pay the liquidator with a corresponding collateral deposit.

Premium Calculation

A premia model is implemented to account for the premium received by options sellers and paid by options buyers. Depending on the fees collected from the deposited liquidity in the

underlying AMM pool, the options sellers can collect the premium when closing their positions. If the deposited liquidity is used by long positions, a spread is added depending on the utilized ratio for the option buyers to pay.

AMM Price Feed

Many operations in the system are sensitive to the price of two assets in the underlying AMM pool. Multiple types of price reads are performed for different operations. For instance, the median of 20 Time-Weighted Average Price (twap) reads from the AMM pool is used to assess liquidation and force exercise eligibility. The protocol itself also maintains a mini-median price to further guard against undesirable effects of potential price manipulation.

Non-voluntary Position Exits

Options positions can be exercised involuntarily in two scenarios. An account becomes liquidatable when the net collateral required from all its active positions exceeds its collateral balance. Anyone can liquidate a margin-called account (i.e., exercising all of its positions) by depositing the required collateral and getting back a bonus on top in collateral shares. Secondly, anyone can force exercise an option position with an out-of-range long leg, thus freeing up corresponding tick-range liquidity with an exercise fee that compensates the option owner.

Security Model and Privileged Roles

The protocol is not upgradeable and the only privileged role is the owner of the Panoptic Factory. The Panoptic Factory owner has the privileged operation of being able to call `updateParameters` on a deployed Panoptic Pool, which in turn calls the function of the same name on each of the pool's Collateral Trackers. This allows the Panoptic Factory owner to update important parameters such as the maintenance margin ratio, commission fee, collateral ratios, pool utilizations and exercise cost.

Disclaimer: *Note that due to substantial refactoring of the Panoptic protocol between its initial audit and the fix review period, it is strongly recommended that the codebase undergoes another audit.*

Critical Severity

C-01 Incorrect Share Accounting When Rolling Options

An options roll can be used to burn a position and transfer the burned position's size to a new OTM position that may differ in only strike and width. Several scenarios can lead to the option owner being able to collect an undue profit or not receiving the profit/loss they are owed due to collateral share accounting during rolls.

When `numeraire` equals `tokenType`

The notional value will be equivalent to `positionSize * optionRatio`, independent of the strike value. When rolling a position to a different strike, the notional values, hence the long/short amounts will be the same. In this case, the net `longAmounts` and `shortAmounts` would be zero. This block will be skipped, thus resulting in zero change (other than premium) in the option owner's share amount. If the original position is in profit or loss, the usual updates to the share balances during exercise will not happen. A more drastic scenario happens when a user first mints an ITM position at a strike price far away from the current tick, which can cause the swapped amount to be small relative to the short (or long) amount. This results in shares minted to the user. They can then roll the position and keep the new shares (which would normally be accounted for during exercise) which are redeemable for profit.

When `numeraire` does not equal `tokenType`

Due to the difference in `numeraire` and `tokenType`, the notional value will be multiplied/divided by the strike price. Thus, rolling a short position to a different strike, a non-zero net `shortAmounts` will be used to account for the intrinsic value. Since the `swappedAmount` in a short option roll comes from burning the old position, with nonzero `shortAmounts`, the exchanged amount will be quite negative too. This will set `tokenToPay` negative and mint shares to the option owner. The owner can amplify their profit by rolling multiple times to get even more shares minted, as rolling does not incur any extra cost. These shares can then be exercised OTM and redeemed for profit.

Consider modifying the share accounting logic during rolls to prevent unaccounted new shares from being minted to the option owner, as well as ensuring that the old position is exercised correctly.

Update: *Resolved at commit 4e1de7a.*

C-02 Roll Token ID Validation Fails

When an `oldTokenId` is rolled into a `newTokenId` from the PanopticPool, the intention inferred from the ROLL_MASK is that only the `width` and `strike` can be different in each leg with all other parameters being the same. However,
when `rolledTokenIsValid` is checked in the SFPM, when the roll validation fails, it will simply skip to the else block that assigns the old and new tokenId without any restriction.

Since the collateral is not checked when rolling an option, if the `newTokenId` has more notional value than the `oldTokenId`, the caller does not need to add more collateral. One way to get a profit from this is to switch the numeraire to get more minted in the "token to pay" logic. For instance, one can make `shortAmounts` of the `newTokenId` significantly bigger by switching the numeraire when the exchange rate between `token0` and `token1` is large. Hence, `tokenToPay` ends up negative which means shares are minted from rolling. One can then burn the position/redeem the shares for a profit.

Consider validating that the new token IDs exactly match the previous ones when a roll is done.

Update: *Resolved at commit 4e1de7a.*

C-03 Shares Can Be Transferred With Open Positions

Shares can be transferred or redeemed from an owner's account while they have open positions. In several cases, `msg.sender` is used where it should be `from` or `owner`:

- `transferFrom`
- `maxWithdraw` used in `withdraw`
- `maxRedeem` used in `redeem`

By approving a second account with no open positions, the owner can thus transfer/redeem their entire share balance with unexercised positions.

- This can expose the Panoptic pool to potential losses when the owner transfers out the shares, leaving under collateralized positions in the pool.
- The owner can collect an undue profit by, for example, minting an ITM position where more shares are minted and then redeeming them.

Consider correcting the logic of the previously mentioned functions to check the number of open positions for the right account.

Update: *Resolved.*

C-04 Liquidation Can Be Avoided Due to Unbounded Position List

Each account in the Panoptic Pool has a `positionIdList` associated with it, containing all active positions. This list is used to validate the input holdings via the positionHash, calculate collateral requirements as well as compute the premia when minting a new `tokenId` or liquidating an entire account. The `positionIdList` is an unbounded array that could potentially expose the Panoptic pool to the following key risks.

Underwater accounts with large `positionIdList` can avoid liquidation

Account liquidation requires the burning of all active options associated with an account. It is possible to avoid liquidation of a margin-called account by keeping a large array of dummy positions to use up the block gas limit. Due to the many loops involved in the `liquidateAccount` call, it takes less than 200 tokenIds to exceed the block gas limit of 30 million.

Well-collateralized account may not be able to mint further

When minting an option, all past positions in `positionIdList` are passed in `mintOptions` as a parameter with the new position to mint appended at the end. The past positions are used to validate the position hash for the `msg.sender` and check the solvency status of the account. When the list of past positions becomes too large, the gas expense of processing the past checks could exceed the block gas limit. This will disable the account to mint a further position. From our experimentation, the limit is approximately of the magnitude of 2000 tokenIds for the current implementation.

In summary, the asymmetry of being able to mint much further than getting liquidated could allow malicious actors to expose the pool to unwanted losses by blocking liquidation. Since it is crucial for the system to evaluate the past positions of an account, consider setting an upper limit on the length of `positionIdList` to ensure that the liquidation of an underwater account can be executed.

Update: *Resolved.*

High Severity

H-01 Undercollateralized ITM Positions Can Be Minted

When minting short options, the collateral requirements are computed as `amount * collateralRatio * maintenanceMarginRatio` where `amount` is the number of options contracts denominated in the numeraire, `collateralRatio` is between 0.2 and 1 depending on the current pool utilization, and `maintenanceMarginRatio` is a fixed quantity (1.3333). The position can be liquidated if the required value is more than the token value of the account balance. These values are calculated from the token data returned by `getAccountMarginDetails`. This function will use an additional term in calculating the required collateral of a position to evaluate the profit/loss of that position when it is ITM, which can result in the required collateral being higher than when minting and can exceed the collateral balance of the user. The user is then immediately vulnerable to liquidation because their position is undercollateralized.

When minting a new ITM position, the profit/loss of the new position is not considered immediately in the collateral calculation, resulting in a position that can be liquidated right away following the mint. Consider calculating collateralization requirements the same way during both minting and when checking if an account is liquidatable.

Update: *Resolved at commit 4e1de7a.*

H-02 Premia Calculation Can Overflow

When calculating the accumulated premia for a position's leg, the accrued fees for the tick range are multiplied by the position's liquidity amount. Both quantities are of type `uint128` and thus can overflow in the intermediate step of `legPremia`. This results in less premium

received by option sellers and less premium paid by option buyers. Additionally, when a position is exercised, this resulting premium value will be passed in and so the locked amount will be updated incorrectly. Consider expanding to 256-bit integers for the leg premia multiplication.

Update: *Resolved. Consider adding a comment to clarify the casting logic of both return values of the* `getAccountPremium` *function from* `uint128` *to* `uint256`*.*

H-03 Leverage Not Available When Tick Is Negative

The available leverage for a newly minted option is based on the current pool utilization. A check is performed comparing the current tick with the median tick, and the utilization will be set to over 100% if there is too much deviation between the ticks. For short positions, this results in `max_sell_ratio` being used as the collateral ratio (i.e., 100%) so the required collateral will be the same as the `shortAmounts`. If the median tick and current tick are both negative and close together (the same, for example), the statement will return true not allowing leverage to be accessible. Consider accounting for negative tick values or checking price deviations instead of tick deviations to correctly set the collateral requirement.

Update: *Resolved.*

Medium Severity

M-01 SFPM Token Transfer Can Fail With Multiple Touches to a Position Key

The position tokenId is minted to the caller as an ERC-1155 token and can be transferred with `safeTransferFrom`. The post-transfer hook will register the token transfer by validating the liquidity of each leg and updating the relevant state variables. For each leg, the `netLiquidity` of the `from` position should be equal to the chunk liquidity, otherwise the transfer will fail. This condition can be violated easily when the same position key is touched more than once, for instance, across multiple legs or across multiple tokenIds. The `fromLiq.rightSlot()` is a net quantity updated by all prior positions while `liquidityChunk.liquidity()` is a leg-specific quantity. Thus, one cannot expect them to

be equal in general. Consider adjusting the required condition to allow legitimate transfer of SFPM tokens.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> SFPM token transfers are only intended as a niche convenience feature primarily for transferring positions to new accounts and are limited by design. Transfers between accounts that share active position keys are not allowed because of security and accounting concerns regarding fee accumulation and integrators (i.e, the Panoptic pool). Legs with duplicate position keys in the same `tokenId` are very unusual and generally not advisable to mint (as an equivalent, more gas-efficient position can always be created by consolidating those legs), and as such we decided not to add extra logic to accomodate the transfer of these positions.

M-02 Delegated Amount Does Not Consider Owed Premia During Force Exercise

An option seller is able to exit a position in the case of insufficient liquidity by force-exercising corresponding out-of-range long positions (from option buyers). The exercise will fail if the buyer's position has accumulated fees in the other token (i.e., not `tokenType`) and there are not enough shares in the buyer's account to pay for it.

More concretely, assume that the long position has `tokenType=1`.

- A buyer can mint a position by having collateral only in the `tokenType` token (i.e., if `tokenType=1`); only deposits in `token1` are needed.
- The long position can accumulate `owedPremia` in both tokens. During exercise, which occurs during the burning of options, the buyer needs to pay the `owedPremia` in both tokens as a multiple of collected fees. This will fail if there are not enough shares in `token0`. Thus, at this stage, the buyer could deposit more `token0` to successfully exit.
- If a seller wants to `forceExercise` this position, the seller needs to transfer to the buyer's account a `delegatedAmounts`, which is the longAmounts in the `tokenType` token (i.e., only in `token1` for this instance). After the delegation, there still isn't enough `token0` to exercise, and the force-exercise will fail.
- This can be mitigated by the seller manually depositing shares to the buyer (set as `receiver`). However, this will cost the seller extra in mevTax.

Consider including `owedPremia` for both tokens in the `delegatedAmounts` during forced exercise.

Update: *Resolved.*

M-03 ETH Can Get Locked in Contracts

There are multiple occurrences throughout the codebase where ETH can become locked. For instance:

- In the `multicall` function of Multicall.sol
- In the `constructor` function of PanopticHelper.sol

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> This is intended. The functions are marked `payable` for a gas cost reduction, and there is no reason for end users to send ETH along with their calls. If they decide to do so, it is equivalent to burning ETH and has no effect on the operations of the protocol.

M-04 Available Assets Do Not Account For Long Positions

When a short position is minted, tokens are moved from the Panoptic Pool to the AMM pool thus reducing the balance of the pool. This amount is reflected in the reduction of the available assets, thus preventing withdrawals.

When a corresponding long position is minted, tokens are moved from the AMM pool to the Panoptic Pool, increasing the balance of the pool. The same amount is then added back to the available assets, allowing any LPs to withdraw their assets when the long position is still active.

With leverage, it is possible that after LP withdrawals, there may not be sufficient liquidity in the Panoptic Pool to transfer to the AMM pool when exercising the long position, thus disabling long positions from closing until further liquidity provision.

Consider accounting for the available assets used by both option buyers and sellers.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> This is intended. Allowing removed long liquidity to be reused for short positions and withdrawals greatly enhances flexibility and user experience in most situations. It is possible for the pool to lack the funds necessary to exercise a long position in certain unusual circumstances, but users who wish to exercise their options have multiple options to free up funds in the event of a liquidity crunch. They can deposit more

> collateral, mint more long positons, close their short positions, or wait for other sellers to close positions.

M-05 Locked Funds May Become Inaccessible

The collected tokens from the AMM trading fees are locked every time a position is touched. In addition, the long position premia payment is also added to the `lockedFunds` when a position is exercised. The outflow of the locked tokens is for short position premia only computed in `s_accountPremiumGross` thus resulting in the possibility of accumulating a positive amount in the locked funds. Further, the calculation of total assets excludes the locked funds, thus withdrawing all shares will not enable access to the locked funds either. This creates the possibility that the locked tokens become inaccessible. Consider adding a way to funnel the locked tokens to the pool to mitigate this scenario.

Update: *Acknowledged, not fixed. The explicit tracking of locked funds was replaced with a system that tracks asset balances in storage. This ignores donations, pending fee payouts, and any other untracked balance changes.*

M-06 Initial Deposit to One Collateral Tracker Could Be Zero

When deploying a new Panoptic pool, the deployer is required to provide a small amount of full-range liquidity to the underlying AMM pool. The returned amounts, `amount0` and `amount1`, are then divided by `100` and deposited into both collateral trackers in the name of the `PanopticFactory` to prevent the ERC-4626 inflation attack. It is possible that one of `amount0` or `amount1` is less than `100`. This happens when the current tick is very close to either `MAX_TICK` or `MIN_TICK` as seen in the `getAmount0Delta` or `getAmount1Delta` formulae. In that case, the initial deposit to one of the collateral trackers could be `0` due to truncation.

While it is generally not possible to obtain enough liquidity to move a pool's tick to `MAX_TICK` or `MIN_TICK`, it is possible for pools with very low liquidity. Depending on the AMM pool, the cost of tick manipulation may be worth the potential profit. This enables the attacker to be the first depositor to the collateral tracker and launch the inflation attack as follows:

- The attacker deposits 1 asset to the collateral tracker to get 1 share back.
- A subsequent LP deposits some amount to the collateral tracker.

- The attacker front-runs the transaction by transferring directly the same amount to the Panoptic pool. Thus, the LP from the previous step would get 0 shares in return due to rounding.
- The attacker can then redeem his single share to get the combined amount of the LP deposit as well as the prior direct transfer.

Since a new Panoptic pool cannot be deployed with the same underlying AMM pool, it would be difficult to stop this attack once the attacker's initial deposit takes place. Consider ensuring that a minimum number of shares are minted in each collateral tracker during pool deployment to be resilient against the inflation attack.

Update: *Resolved*.

M-07 Liquidator Can Incur Loss During Liquidation

An account can be liquidated if it does not have enough collateral to cover its position(s), which is an important operation to ensure the health of the protocol. Administrating the insolvent account proceeds by first having the liquidator delegate an amount of shares to cover the account's position(s), then exercising the entire position list of the account, and finally returning the delegated amount plus a bonus to the liquidator. It's possible for a liquidator to lose money because when all positions are exercised, shares can be minted which increases the total supply of shares. This can result in the asset value of the delegated shares plus the bonus after the liquidation being less than the asset value of the delegated shares before the liquidation due to dilution of the share value. Consider ensuring that performing a liquidation always results in a net gain for the liquidator in order for there to be an incentive to perform the operation.

Update: *Resolved at commit 02cd20d.*

Low Severity

L-01 Slippage Protection Should Allow Specifying Exact Tick

When minting options, a check is made where the current tick must be between the specified limits. If the upper limit and lower limit are equal, then the check passes regardless of where the current tick is. A minter may only desire to mint at a specific tick, and may assume this will happen if they set `tickLimitLow == tickLimitHigh`. Instead, this currently implies no slippage protection. Considering changing the logic of `_getPriceAndCheckSlippageViolation` to allow minters to specify a specific tick.

Update: *Resolved.*

L-02 Minting Fails When Width and Tick Spacing Are Both Equal to 1

When creating a `tokenId`, specifying a `width` equal to 1 and the underlying pool has a `tickSpacing` of 1, a rounding to zero occurs and the resulting lower and upper tick will be the same as the strike. This will eventually fail when the position is minted in Uniswap. Consider reverting with an error in `asTicks` to have more consistent behavior.

Update: *Resolved in pull request #633 at commit 2fcae1e.*

L-03 `startPool` Can Be Frontrun to Incorrectly Set `medianTick`

When a pool is started, the current tick is read from the Uniswap pool to assign to `miniMedian`. It is possible to front-run `startPool` and manipulate the current tick, affecting the `medianTick` for operations such as minting options (attacker could mint options at a discount for example). Consider adding additional logic for slippage protection when starting a pool.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> If a pool is initialized with an out-of-sync median tick, it is up to the consumers of that pool not to interact until the tick is back in sync. This can be acheived by calling the `poke` function several times. Circumscribing the available current ticks that a pool can be created at does not guarantee protection against median tick manipulation and could cause difficulties when creating smaller pools. Users should decide whether or not to use a certain pool based on their personal risk criteria and preferences.

L-04 Assumptions on Validation Time With `block.number`

The mini-Median can be updated if at least 4 blocks have passed since the last update. This is supposed to correspond to approximately 1 minute (~13-second average validation time on Ethereum) according to the comment, but this may not always be the case:

- Ethereum upgrades may change the validation time.
- Deploying the protocol on other chains (L2s like Arbitrum for example), 4 blocks can take only several seconds.

Consider using `block.timestamp` instead of `block.number` for the mini-Median to more precisely ensure the desired time interval has passed. Current epoch time takes up about 30 bits to store as an integer so the 40 bits used to store the block information in `s_miniMedian` would be sufficient going forward.

Update: *Resolved in pull request #2 at commit 3cd6bd6.*

L-05 Writing Over Existing `tokenId` May Overflow Certain Assigned Bits

The `TokenId` library contains multiple helper functions to update specific assigned bits in a `tokenId`. In the case of overwriting an existing leg, there is a risk of overflowing a single range of bits. For example, if the method `addIsLong` adds a bit value of `1` to an existing `isLong` value of `1`, the leftmost bit will overflow to the higher-order bit. This will flip the value of `tokenType`, changing the user's option type.

Consider adding overflow detection checks to prevent overwriting untargeted bit-constituents of the `tokenId`.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> This is intended behavior. These functions are designed to be as gas-efficient as possible, and as such it is expected that user inputs will be validated before they are called. Specifically, they are only intended to be used on token ids with the respective slots cleared (such as when one is being built starting from `uint256(0)`). They are prefixed with `add` for a reason - if the functions cleared slots first and did overflow checks they would be prefixed with `set` instead.

L-06 Missing Error Messages in `revert` Statements

Throughout the codebase, there are `revert` statements that lack error messages. For instance:

- The `revert` statement within the `poolData` function on line 409 of `PanopticPool.sol`
- The `revert` statement within the `convertToTokenValue` function on line 327 of `PanopticMath.sol`

Consider including specific, informative error messages in `revert` statements to improve the overall clarity of the codebase and to avoid potential confusion when the contract reverts.

Update: *Partially fixed. The Panoptic team stated:*

> There is not a specific commit for this, but a search of the latest version of the contracts will find that these empty revert statements are no longer present. There are only two possible empty reverts, both found in assembly blocks in the `mulDivDown` and `mulDivUp` functions taken from Solmate and used in the `CollateralTracker`'s ERC-4626 implementation. Solmate's standard ERC-4626 also reverts with no data on overflows, and the same behavior is kept here for consistency.

L-07 `legIndex` Is Not Modulo Four

When updating parameters in a tokenId (for example, in `addNumeraire`), a leg index can be specified which should be at most 3. During the left shift operation, the leg index is not taken modulo 4 and so providing a value greater than 3 will shift by more than 256 bits and will result in no change to the tokenId. This is inconsistent with how other parameters are handled which are always taken modulo their maximum value during the shift, for example, when updating the option ratio, token type, etc. Consider taking the leg index modulo four to remain consistent with other operations.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> If a leg index greater than 3 is specified, the expected behavior is that the `tokenId` should not change because there are only 4 legs, so updating a fifth leg would have no effect. The leg parameters are constricted to their maximum values because a value greater than the maximum would be nonsensical and bleed into the other slots in the `tokenId`, but it is reasonable to expect that attempting to update a fifth leg would leave the `tokenId` unchanged and is therefore harmless.

L-08 Arguments Passed to Functions in Reverse Order

During the creation of some options strategies in `PanopticHelper`, the arguments passed to underlying functions are passed in a reversed order. This will cause users to set the wrong `numeraire` as well as the wrong leg type when creating options strategies. For instance:

- In `createJadeLizard`, the arguments `numeraire` and `isLong` are passed to `createStrangle` in a reversed order.
- In `createBigLizard`, the arguments `numeraire` and `isLong` are passed to `createStraddle` in a reversed order.

Consider passing the arguments `numeraire` and `isLong` in the correct order in the above-mentioned functions.

Update: *Resolved at commit 2f2addf.*

L-09 Casting Risk in Premia Values

Values packed into the right and left slots of `legPremia` (type `int256`) are unsigned 128-bit integers. Those values will be cast to `int128` when written and then subsequently read as `int128`. If the original unsigned values have a non-zero most-significant bit then the result will be read incorrectly as a negative number. Consider verifying that all values cast to a signed integer are equal to their original unsigned values.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> The protocol, for multiple reasons, makes the assumption that the premia will not exceed (2**127-1). The maximum feasible collateral that can be deposited is orders of magnitude smaller than that, and it is inconcievable that an amount of premium that

> large on any legitimate pool could be accumulated. Adding a safecast here would not resolve issues stemming from an extremely large amount of premium being accumulated and would also cause positions to get stuck.

L-10 Considerations Regarding AMM TWAP Readings

The AMM `twapTick` is used in `forceExercise` to determine if a `tokenId` is forcibly exercisable. It is also used in `liquidateAccount` to compute the collateral required to determine if a said `tokenId` is margin called. The `twapTick` is constructed by taking the median of 20 observations from the AMM pool at intervals of 30s.

Depending on the observation arrays from different AMM pools, the following scenarios may be worth considering.

When the oldest observation is too recent

The TWAP read will revert if the oldest observation is too recent (i.e., less than 600s). This happens when the observation array cardinality is small. For instance, at the time of writing, the DAI-WETH-100bps pool has an observation cardinality of 1, thus each new observation overwrites the current one. Hence, one can easily write a new observation via swapping in the AMM pool to DoS the TWAP read.

This is significantly mitigated in the deployment of the Panoptic pool as the observation cardinality is increased to a minimum of 100. However, this happens after `_mintFullRange`, which writes a new observation to the array before expanding it to 100. Thus, there may still be a brief window where TWAP reads can revert.

Consider putting `increaseObservationCardinalityNext` before `_mintFullRange`, as this will fill up one extra slot after expanding.

When the latest observation is older than TWAP_WINDOW

If the latest observation from the AMM pool was updated more than TWAP_WINDOW seconds ago, then the twapMeasurement will always be the `currentTick`. Thus by checking the timestamp of the latest observation, one could avoid the gas-intensive reads from the AMM.

In both small and large AMM pools, observations with gaps larger than 600s can happen more frequently than assumed. It would be good to consider the appropriateness of the TWAP_WINDOW length across different pools.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> It is unusual for the latest observation to be older than the TWAP window except on very inactive pools. While it is possible to reduce gas in that situation by computing the TWAP with simpler logic, it is not common enough to warrant its own optimized branch. If it is common on a certain pool, it may be advisable for that pool to be deployed with a longer TWAP window.

L-11 Potential Division by Zero When Checking Liquidity Spread

The function `_checkLiquiditySpread` ensures that the effective liquidity in a given chunk is above a certain threshold. However, when the liquidity spread is computed, the function will revert when `netLiquidity` is equal to zero. This can happen if a user tries to open a long position with all the available liquidity.

Consider preventing the division when `netLiquidity` is zero and reverting with an informative and user-friendly error message.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> The Panoptic Pool enforces a maximum spread multiplier, meaning that at least 10% of a given liquidity chunk must remain in the AMM at all times. Therefore, `netLiquidity` will never be zero when this calculation is performed.

L-12 Straddle Legs Are Not Risk-Partnered

In `PanopticHelper.sol`, the function `createStraddle` creates a call and a put leg with identical strike prices. Both the call and the put legs are expected to have each other assigned as risk partners, as described in this comment.

However, when creating the call leg and the put leg, each leg gets assigned as its own risk partner.

Consider partnering the call and put legs together.

Update: *Resolved at commit 2f2addf.*

L-13 Misleading or Incorrect Docstrings

There are some instances in the codebase where the comments are misleading or can be improved for clarity.

- When the parameter `collateralCalculation` is `true`, the premium will be computed for all options regardless if `isLong` is `0` or `1`, whereas the comment states that `if true do not compute premium of short options`.
- The `v` in the formula derivation of Eqn 2 is in the wrong place. It should be `gross_feesCollectedX128 = feeGrowthX128*N + feeGrowthX128*S*(1 + v*S/N)` instead.
- The variable name in Eqn 3 should be `s_accountPremiumOwed` instead of `s_accountLiquidityOwed`.
- The account premium variables have 64 bits precision but these are not mentioned in the doc-string. This may cause confusion.
- The comment above `s_miniMedian` should say the block number occupies the most significant 40 bits (not 32), and there are 8 interactions stored (the diagram shown starts from index 1 which would mean 7 interactions in total)
- The comment for `mintTokenizedPosition` says that the function reverts if the position is not unique. However, it is possible to call the function twice with the same exact input.
- When swapping 0 for 1 the price actually moves downwards (the price is token1/token0, token0 is added to the pool)
- In `CollateralTracker.sol`, this comment has a typo. "`minNum(Half)RangesFromStrike`" should be "`maxNum(Half)RangesFromStrike`".
- In `TokenId.sol`, this comment has a typo. "incoming 16 bits" should be "incoming 24 bits".

Update: *Resolved at commit 993f4b5.*

Notes & Additional Information

N-01 Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function body for the purpose of being returned as the function's output. They are an alternative to explicit in-line `return` statements.

Throughout the codebase, there are multiple instances of unused named return variables.

For instance:

- The `_symbol` return variable in the `symbol` function in `CollateralTracker.sol`.
- The `assetTokenAddress` return variable in the `asset` function in `CollateralTracker.sol`.
- The `totalManagedAssets` return variable in the `totalAssets` function in `CollateralTracker.sol`.
- The `shares` return variable in the `convertToShares` function in `CollateralTracker.sol`.
- The `assets` return variable in the `convertToAssets` function in `CollateralTracker.sol`.
- The `maxAssets` return variable in the `maxDeposit` function in `CollateralTracker.sol`.
- The `shares` return variable in the `previewDeposit` function in `CollateralTracker.sol`.
- The `maxShares` return variable in the `maxMint` function in `CollateralTracker.sol`.
- The `assets` return variable in the `previewMint` function in `CollateralTracker.sol`.
- The `maxAssets` return variable in the `maxWithdraw` function in `CollateralTracker.sol`.
- The `shares` return variable in the `previewWithdraw` function in `CollateralTracker.sol`.
- The `maxShares` return variable in the `maxRedeem` function in `CollateralTracker.sol`.

- The `assets` return variable in the `previewRedeem` function in `CollateralTracker.sol`.
- The `required` return variable in the `_getRequiredCollateralSingleLeg` function in `CollateralTracker.sol`.
- The `panopticPoolBalance` return variable in the `poolData` function in `PanopticPool.sol`.
- The `totalBalance` return variable in the `poolData` function in `PanopticPool.sol`.
- The `inAMM` return variable in the `poolData` function in `PanopticPool.sol`.
- The `totalLocked` return variable in the `poolData` function in `PanopticPool.sol`.
- The `currentPoolUtilization` return variable in the `poolData` function in `PanopticPool.sol`.
- The `premium0` return variable in the `calculateAccumulatedFeesBatch` function in `PanopticPool.sol`.
- The `premium1` return variable in the `calculateAccumulatedFeesBatch` function in `PanopticPool.sol`.
- The `liquidationTick` return variable in the `findLiquidationPriceDown` function in `PanopticHelper.sol`.
- The `liquidationTick` return variable in the `findLiquidationPriceUp` function in `PanopticHelper.sol`.

Consider either using or removing any unused named return variables.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> We use named return variables deliberately in various areas of the codebase to help with comprehension. They are sometimes used throughout the body of the function (as accumulators, for example) but are otherwise intended to assist with code comprehension, even if they are unused.

N-02 Unused Imports

Throughout the codebase, there are imports that are unused and could be removed. For instance:

- Import `FullMath` of `PanopticFactory.sol`
- Import `PeripheryImmutableState` of `PanopticFactory.sol`
- Import `FixedPoint96` of `PanopticPool.sol`
- Import `PoolAddress` of `INonfungiblePositionManager.sol`

- Import `FixedPoint128` of `FeesCalc.sol`
- Import `FullMath` of `FeesCalc.sol`
- Import `SqrtPriceMath` of `FeesCalc.sol`
- Import `Errors` of `FeesCalc.sol`
- Import `TickMath` of `LiquidityChunk.sol`
- Import `TokenId` of `LiquidityChunk.sol`
- Import `FixedPoint96` of `PanopticMath.sol`
- Import `FixedPoint128` of `PanopticMath.sol`
- Import `FullMath` of `PanopticMath.sol`
- Import `SqrtPriceMath` of `PanopticMath.sol`
- Import `Errors` of `PanopticMath.sol`
- Import `TickMath` of `TickPriceFeeInfo.sol`
- Import `TokenId` of `TickPriceFeeInfo.sol`
- Import `PanopticMath` of `PanopticHelper.sol`
- Import `PanopticMath` of `PanopticMigrator.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved at commit 993f4b5.*

N-03 State Variable Visibility Not Explicitly Declared

Throughout the codebase, there are state variables that lack an explicitly declared visibility. For instance:

- The state variable SFPM in PanopticHelper.sol
- The state variable DUST_THRESHOLD in PanopticMigrator.sol

For clarity, consider always explicitly declaring the visibility of variables, even when the default visibility matches the intended visibility.

Update: *Resolved at commit 993f4b5.*

N-04 Identical Custom Errors Used in Different Contexts

Within `SemiFungiblePositionManager.sol`, the custom error `NotEnoughLiquidity()` is used when there is not enough `netLiquidity` for a long position. It is also used in a different context when the specified liquidity amount for a position is lower than the `DUST_THRESHOLD`. Consider using specific, informative custom errors in different contexts to improve overall code clarity and to facilitate troubleshooting whenever a requirement is not satisfied.

Update: *Resolved.*

N-05 Non-negative Values As Signed Integers In External Interface

Some variables are supposed to take only non-negative values but appear as signed integers in the interface of a function (i.e., either in the arguments or return variables).

For instance, the `width` of a leg is always positive but appears to be `int24`. Thus it is possible to input a negative width in the option strategies, for example in `createStrangle` to create a valid but unexpected `tokenId`. An input of `width = -2` gives an output of `width = 4094`, thus passing the `validate()` function, giving unexpected results.

Other examples include non-negative values for `poolUtilization` as LeftRight `int128` return variable and non-negative `bonusAmounts` as LeftRight `int256`. These may be prone to casting issues for external integration in the future.

Consider keeping non-negative variables as unsigned integers in the external interface.

Update: *Acknowledged, not fixed. The Panoptic team stated:*

> In certain situations it makes sense to represent values which can only be natural numbers as signed integers -- these values are often operated upon with and combined with values that must be signed, so making the types similar often reduces casting headaches and footguns.

Conclusions

Several critical and high-severity issues were found among various medium to lower-severity issues. Given the complexity of the system and the number of issues raised, we suspect there could be more undiscovered issues. In the case of any significant change to the codebase, we recommend another round of auditing on the new code.

Monitoring Recommendations

While audits help in identifying potential security risks, the Panoptic team is encouraged to also incorporate automated monitoring of on-chain contract activity into their operations. Ongoing monitoring of deployed contracts helps in identifying potential threats and issues affecting the production environment.

Here are some example scenarios that would benefit from monitoring:

- **Monitor large price changes in underlying AMM pools**: Positions can become profitable, liquidatable or forcibly exercisable depending on the underlying AMM pool price changes. It would be helpful to monitor this which could be the impetus for increased options activities on Panoptic pools.
- **Monitor underwater accounts**: Accounts can slide into a liquidatable state when the TWAP price changes unfavorably and the collateral deposits cannot cover the loss of their positions. When underwater accounts do not get liquidated in a timely fashion, it is helpful for the protocol to intervene and preserve the integrity of the system.
- **Monitor privileged role activities**: Privileged role activities such as those performed by the factory owner should be monitored to detect compromised private keys.

Disclaimer: *Note that fixes are distributed among two different repos, panoptic-labs/Panoptic and panoptic-labs/panoptic-v1-core.*