Panoptic Hypovault Audit Report

Version 1.0

Starkxun

August 12, 2025

Panoptic Hypovault Audit Report

Starkxun

August 12, 2025

Prepared by: Starkxun Lead Auditors: - Starkxun

Table of Contents

Protocol Summary

Panoptic's HypoVault is a liquidity vault contract designed to integrate with the Panoptic protocol's perpetual options trading system. The vault accepts deposits of a single underlying ERC-20 token from liquidity providers (LPs), and uses the pooled assets to provide concentrated liquidity in Uniswap v3 pools that are utilized for Panoptic's options positions.

HypoVault automates several functions for LPs:

- **Deposit & Redemption** — Users deposit the underlying token and receive vault shares; shares can be redeemed for the proportional underlying plus accrued trading premiums.
- **Performance Fee Handling** — The vault deducts performance fees from profits according to predefined parameters, distributing them to the fee recipient.
- **NAV (Net Asset Value) Calculation** — Computes vault value as underlying balance plus the converted value of accrued premiums in other tokens.
- **Risk & Exposure Management** — Uses on-chain price feeds and position data to track liquidity exposure and manage rebalancing.
- **Integration with Panoptic Positions** — Liquidity is used as collateral for Panoptic's perpetual options market-making, aiming to generate yield from option premiums and trading activity.

The HypoVault contract is an essential component of Panoptic's DeFi options architecture, enabling passive LPs to participate in the Panoptic ecosystem without managing individual Uniswap positions manually.

Disclaimer

This audit was conducted by an independent security researcher (Starkxun) without affiliation to the Panoptic team. Every effort has been made to identify potential vulnerabilities in the reviewed code during the allocated time frame. However, no security review can guarantee the complete absence of flaws.

This report does not constitute an endorsement of the protocol's underlying business model, tokenomics, or legal compliance. The audit covered only the Solidity smart contract code specified in the agreed scope. Any changes made after the audit period are not covered by this report and may introduce additional risks. Users and stakeholders should perform their own due diligence before engaging with the protocol.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The scope of this audit covered the Solidity smart contracts related to the Panoptic HypoVault implementation. The reviewed commit hash: 8ef6d867a5fb6ffd1a6cc479a2380a611d452b4a Repository: Panoptic HypoVault C4 Repo

Files in scope:

- src/HypoVault.sol
- src/accountants/PanopticVaultAccountant.sol

Out of scope:

- src/interfaces/IVaultAccountant.sol
- test/.

Roles

Role	Description	Key Permissions
Depositor	Any user depositing the underlying asset into the vault to receive shares.	Deposit underlying tokens, redeem shares.
Vault Owner	Account or contract controlling vault parameters.	Configure fees, set fee recipient, trigger certain vault operations.
Fee Recipient	Address receiving the vault's performance fees.	No direct control over funds, only receives fee distributions.
Keeper / Rebalancer	Off-chain or on-chain agent performing liquidity rebalancing and maintenance.	May trigger rebalancing or position updates, depending on access controls.

Executive Summary

Issues found

Severity	Number of issues found
High	1
Medium	1
Low	0
Info	0
Total	2

Findings

High

[H-1] Incorrect `poolExposure1` Calculation Causes NAV Underestimation

Description: In `PanopticVaultAccountant.computeNAV()`, the exposure for `token1` (`poolExposure1`) is calculated incorrectly as:

```
poolExposure1 = int256(uint256(longPremium.leftSlot())) - int256(
    uint256(shortPremium.leftSlot()));
```

This reverses the intended formula and effectively computes `long - short` instead of `short - long`.

In Panoptic's premium model:

- shortPremium represents accrued premiums owed to short positions (asset for the vault, should increase NAV).
- longPremium represents accrued premiums owed to long positions (liability for the vault, should decrease NAV).

Therefore, correct exposure for each token should be:

```
poolExposure = shortPremium - longPremium
```

However, the incorrect calculation for token1 leads to a negative net exposure, underestimating the vault's NAV.

Impact:

Due to the reversed calculation for token1, NAV is underestimated. This directly impacts:

- Deposits: Users may receive more shares than they should.
- Withdrawals: Users may receive fewer assets than they are entitled to.

Over time, this creates a value mismatch between shares and underlying assets, harming vault balance integrity.

Proof of Concept:

Proof Of Code

```
        function test_submissionValidity() public {
        // Create pool where token0 is the underlying token, only token1 needs conversion
        PanopticVaultAccountant.PoolInfo[] memory pools = new PanopticVaultAccountant.PoolInfo[](1);
        pools[0] = PanopticVaultAccountant.PoolInfo({
            pool: PanopticPool(address(mockPool)),
            token0: underlyingToken, // Same as underlying
            token1: token1,
            poolOracle: poolOracle,
            oracle0: oracle0,
            isUnderlyingToken0InOracle0: true,
            oracle1: oracle1,
            isUnderlyingToken0InOracle1: false,
            maxPriceDeviation: MAX_PRICE_DEVIATION,
            twapWindow: TWAP_WINDOW
        });

        // Update vault's pools hash (use address(vault) as vault's address))
        accountant.updatePoolsHash(address(vault), keccak256(abi.encode(pools)));

         // Setup scenario: no token balances, only underlying balance and premiums
        underlyingToken.setBalance(address(vault), 1000e18);
        token1.setBalance(address(vault), 0); // No token1 balance
        mockPool.collateralToken0().setBalance(address(vault), 0); // No collateral
        mockPool.collateralToken0().setPreviewRedeemReturn(0);
        mockPool.collateralToken1().setBalance(address(vault), 0);
        mockPool.collateralToken1().setPreviewRedeemReturn(0);

        uint256 shortPremiumRight = 200e18;
        uint256 shortPremiumLeft = 150e18;
        uint256 longPremiumRight = 50e18;
        uint256 longPremiumLeft = 50e18;

        uint256 net = (shortPremiumRight - longPremiumRight) + (shortPremiumLeft - longPremiumLeft);
        assertEq(net, 250e18, "Net premium should be 250 ether");

        mockPool.setMockPremiums(
            LeftRightUnsigned.wrap((shortPremiumLeft << 128) | shortPremiumRight),
            LeftRightUnsigned.wrap((longPremiumLeft << 128) | longPremiumRight)
        );

        // No positions
        mockPool.setNumberOfLegs(address(vault), 0);
        mockPool.setMockPositionBalanceArray(new uint256[2][](0));

        bytes memory managerInput = createManagerInput(pools, new TokenId[][](1));

        uint256 nav = accountant.computeNAV(address(vault), address(underlyingToken), managerInput);

        // expect NAV:
        // expectedNavBase = underlyingToken + net = 1000e18 + 250e18 = 1250e18
        uint256 expectedNavBase = 1000e18 + net; // Conservative estimate
        uint256 tolerance = 10e18; // Small tolerance for premium conversion calculations

        assertApproxEqAbs(
            nav,
            expectedNavBase,
            tolerance,
            "NAV should include underlying plus converted premiums"
        );
    }

    function createManagerInput(
        PanopticVaultAccountant.PoolInfo[] memory pools,
        TokenId[][] memory tokenIds
    ) internal pure returns (bytes memory) {
        PanopticVaultAccountant.ManagerPrices[]
            memory managerPrices = new PanopticVaultAccountant.
               ManagerPrices[](pools.length);

        for (uint i = 0; i < pools.length; i++) {
            managerPrices[i] = PanopticVaultAccountant.ManagerPrices({
                poolPrice: TWAP_TICK,
                token0Price: TWAP_TICK,
                token1Price: TWAP_TICK
            });
        }

        return abi.encode(managerPrices, pools, tokenIds);
    }
```

Recommended Mitigation: Update poolExposure1 calculation to match poolExposure0 logic:

```
    {

        LeftRightUnsigned shortPremium;
        LeftRightUnsigned longPremium;

        (shortPremium, longPremium, positionBalanceArray) = pools[i]
            .pool
            .getAccumulatedFeesAndPositionsData(_vault, true, tokenIds[
               i]);


        poolExposure0 =
            int256(uint256(shortPremium.rightSlot())) -
            int256(uint256(longPremium.rightSlot()));


-        poolExposure1 =
-            int256(uint256(longPremium.leftSlot())) -
-            int256(uint256(shortPremium.leftSlot()));

+        poolExposure1 =
+            int256(uint256(shortPremium.leftSlot())) -
+            int256(uint256(longPremium.leftSlot()));
```

```

    }
```

Medium

[M-1] NAV Miscalculation Due to Underlying Token Being Added After Pool Exposure Truncation

Description: In the `PanopticVaultAccountant::computeNAV()` function, the vault's underlying token balance is added to the NAV after per-pool exposures have already been truncated using `Math.max(poolExposure0 + poolExposure1, 0)`.

This logic leads to inconsistent NAV calculation when the underlying token is not part of any pool: negative exposure is zeroed out, and then the token balance is added directly — resulting in an overestimated NAV.

Impact: Vaults with economically identical exposures but different pool configurations (i.e., whether or not they contain the underlying token) will report different NAV values.

This can: - Cause incorrect share issuance during deposits, - Lead to unfair share redemptions during withdrawals, - Create opportunities for economic exploitation due to inaccurate vault valuation.

Proof of Concept:

Proof Of Code

```
    function test_computeNAV_negativeExposure_inconsistency() public {
        // Pool: ETH/WBTC, underlying: USDC (not in pool)
        // Simulate exposure = -150 USDC, vault balance = 50 USDC

        PanopticVaultAccountant.PoolInfo ;
        pools[0] = PanopticVaultAccountant.PoolInfo({
            pool: PanopticPool(address(mockPool)),
            token0: token0, // ETH
            token1: token1, // WBTC
            poolOracle: poolOracle,
            oracle0: oracle0,
            isUnderlyingToken0InOracle0: false,
            oracle1: oracle1,
            isUnderlyingToken0InOracle1: false,
            maxPriceDeviation: MAX_PRICE_DEVIATION,
            twapWindow: TWAP_WINDOW
        });

        accountant.updatePoolsHash(vault, keccak256(abi.encode(pools)));

        underlyingToken.setBalance(vault, 50e18);  // +50 USDC balance
        token0.setBalance(vault, 0);                  // 0 ETH
        token1.setBalance(vault, 0);                  // 0 WBTC
        mockPool.collateralToken0().setBalance(vault, 0);
        mockPool.collateralToken1().setBalance(vault, 0);

        // Exposure = -150 USDC via premiums
        mockPool.setMockPremiums(
            LeftRightUnsigned.wrap(0),               // no short premium
            LeftRightUnsigned.wrap(150e18)           // long premium = 150 USDC
        );

        mockPool.setNumberOfLegs(vault, 0);
        mockPool.setMockPositionBalanceArray(new uint256 );

        PanopticVaultAccountant.ManagerPrices ;
        managerPrices[0] = PanopticVaultAccountant.ManagerPrices({
            poolPrice: TWAP_TICK,
            token0Price: TWAP_TICK,
            token1Price: TWAP_TICK
        });

        bytes memory managerInput = abi.encode(managerPrices, pools, new TokenId );

        uint256 nav = accountant.computeNAV(vault, address(underlyingToken), managerInput);

        // Current (buggy) behavior: NAV = 50 (incorrect)
        assertEq(nav, 50e18, "NAV should incorrectly include underlying balance (50 USDC)");

        // Expected correct behavior: NAV = 0
        assertEq(nav, 0, "NAV should be 0, but the bug causes incorrect inclusion of balance");
    }
```

Recommended Mitigation: Postpone the truncation logic (Math.max) until all exposures — including the vault's underlying token balance — have been summed together.

```
function computeNAV(address vault, address underlying, bytes memory managerInput) public view returns (uint256) {
    // ... Other Logic ...
    uint256 nav = 0;
    for (uint i = 0; i < pools.length; i++) {
        // ...  Other Logic ...

        // debt in pools with negative exposure does not need to be paid back
-       nav += uint256(Math.max(poolExposure0 + poolExposure1, 0));
+       nav += uint256(poolExposure0 + poolExposure1);
```

```
    }

    bool skipUnderlying = false;
    for (uint256 i = 0; i < underlyingTokens.length; i++) {
        if (underlyingTokens[i] == underlyingToken) skipUnderlying = true;
    }
    if (!skipUnderlying) nav += IERC20Partial(underlyingToken).balanceOf(_vault);

+    nav = uint256(Math.max(int256(nav), 0));
```

This ensures that negative exposure is not improperly offset by underlying balance.

Low

Informational

Gas