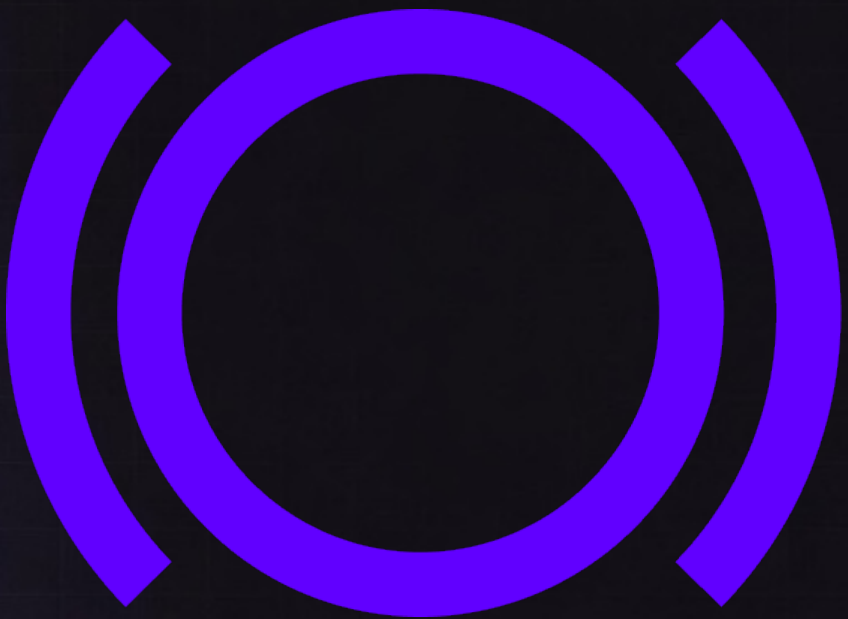
# Panoptic

Security Review



March 20, 2026

**Prepared by:** Valves Security

**Lead Auditors:** Vesko210 | Merulez99

# Contents

# 1 About Valves Security

Valves Security is a Web3 security team founded by Veselin and Valeri. With deep experience across EVM-based protocols, Cosmos (Go) chains, and native Rust systems, we help projects identify and eliminate critical vulnerabilities. With a strong background and a strong team of freelancers we can asure a top quality audit.

Book a Security Review at valvessecurity.com or message us on X Valves Security

# 2 Disclaimer

The Valves team makes all effort to find as many vulnerabilities in the code in the given time period. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the implementation. We stand by our process, yet encourage follow-up audits and meaningful bug bounty incentives for continued whitehat protection.

# 3 Risk Classification

| | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: High** | **Critical** | **High** | **Medium** |
| **Likelihood: Medium** | **High** | **Medium** | **Low** |
| **Likelihood: Low** | **Medium** | **Low** | **Low** |

### Impact

- High - leads to a significant material loss of assets in the protocol or a group of users
- Medium - leads to a moderate material loss of assets in the protocol, moderately harms a group of users or to some disruption of the protocol's functionality
- Low - No funds at risk

### Likelihood

- High - attack path is possible with reasonable assumptions. The issue is almost certain to happen.
- Medium - only a conditionally incentivized attack vector, but still relatively likely

- Low - has unlikely assumptions or requires a significant stake by the attacker

## 4 Protocol Summary

Panoptic is a decentralized derivatives protocol that introduces perpetual, oracle-free options on top of existing AMM liquidity. By leveraging Uniswap v3 liquidity positions as the foundation for options, the protocol enables permissionless minting, trading, and market-making of put and call options on any token pair-without intermediaries, expiries, or external price feeds. This design transforms liquidity into programmable volatility exposure, creating a capital-efficient, composable options primitive that aligns on-chain pricing with real market activity.

## 5 Executive Summary

### Overview

| Project | Panoptic |
|---|---|
| Commit Hash | dd51c7f0bdb0016c18dd93b61c1e48aaae60c204 |
| Timeline | 17/03/2026 – 20/03/2026 |

### Scope

| Id | Files in scope |
|---|---|
| 1 | PanopticVaultAccountant.sol, HypoVault.sol, HypoVaultFactory.sol |

### Issues Found

| | |
|---|---|
| Critical Risk | 0 |
| High Risk | 1 |
| Medium Risk | 0 |
| Low/Info Risk | 4 |

# 6 Findings

## High Findings

### [H-01] Inconsistent spot/TWAP pricing basis in computeNAV() makes NAV inaccurate

**Fixed**

**Description**

`PanopticVaultAccountant.computeNAV()` uses two different price bases within the same valuation path.

For non-credit legs (width > 0), the accountant derives the token0/token1 composition of each liquidity chunk using the pool's current spot tick:

```
1  pool.getCurrentTick() -> Math.getAmountsForLiquidity(...)
```

However, later in the same function, the resulting exposures are converted into the vault's underlyingToken using the pool TWAP obtained from pool.getTWAP().

As a result, the accountant is not performing a coherent valuation under a single price basis. It composes the position using spot, but values that composition using TWAP. This creates a manipulable pricing surface during the only two lifecycle points where NAV matters for user settlement:

- `fulfillDeposits()`, where NAV determines how many shares depositors receive
- `fulfillWithdrawals()`, where NAV determines how many assets withdrawing users receive

An attacker can exploit this by sandwiching the manager's `fulfillDeposits()` or `fulfillWithdrawals()` transaction. By moving the pool spot tick immediately before the fulfillment transaction executes, the attacker can distort the token0/token1 split returned by `getAmountsForLiquidity()`. If the TWAP deviation checks still pass, the accountant will accept the distorted composition and convert it using TWAP, producing an inaccurate NAV.

The issue is not that spot is used, or that TWAP is used, but that both are used simultaneously for different parts of the same valuation flow.

**Attack scenario**

**Setup:** Vault holds a `LINK/UNI` Panoptic pool position. Vault has deposited 100 LINK as collateral into `collateralToken0`. Manager is preparing to call `fulfillDeposits()` to settle pending depositors.

**Block N - Attacker front-runs:**

1. Attacker executes a large swap on the LINK/UNI Uniswap pool - sells massive LINK, buying UNI
2. Spot crashes: tick 23,027 → 13,863 (LINK drops from 10 UNI to 4 UNI)
3. TWAP barely moves: still ~22,885 (EMA hasn't updated yet - sameblock)

Block N - Manager's `fulfillDeposits()` executes (same block, next tx):

4. Manager reads `pool.getTWAP()` = 22,885 → submits this as poolPrice in `managerInput`
5. computeNAV() runs:

- `pool.getCurrentTick()` = 13,863 (spot, crashed)

- `getAmountsForLiquidity(13863, chunk)` → position is below range [18000, 26760] → all token0 (LINK)
- `previewRedeem()` → 100 LINK collateral
- `pool.getTWAP()` = 22,885 → stale
- Deviation check: |22885 - 22885| = 0 ≤ 1000 → passes
- Conversion: 100+ LINK × TWAP_price (9.86 UNI/LINK) = ~986 UNI reported
- Real value: 100 LINK × spot (4 UNI/LINK) = ~400 UNI actual

6. Depositors in this batch receive shares based on NAV = 986 UNI - 2.46× fewer shares than they should

**Block N - Attacker back-runs:**

7. Attacker swaps back, restoring spot price and collecting the Uniswap fee arbitrage profit
   Why the deviation check doesn't stop this:
   The check at L295 is `|managerPrice - twapTick|`. The manager honestly reads TWAP (22,885) and submits it. `TWAP` is also 22,885. Deviation = 0. The check was designed to prevent a malicious manager from submitting an arbitrary price - it was not designed to detect TWAP staleness relative to spot. There is no `|getCurrentTick() - twapTick|` check anywhere in the function.

Profit source for the attacker: If the attacker is a withdrawer (not a depositor), the inflated NAV also means fulfillWithdrawals() pays them more UNI per share than the vault is actually worth. In the inflated-NAV direction, the attacker can front-run a withdrawal batch instead, receive excess UNI, then back-run to restore the price - extracting value directly from the vault's assets.

**Recommended mitigation** Use a single pricing basis throughout NAV computation.

Use `twapTick` as the single price source for both amount computation and conversion. Replace the `getCurrentTick()` call at L199 with `managerPrices[i].poolPrice`, which is already validated against TWAP by the `StaleOraclePrice` guard:

If the intended design is to keep spot-based composition, then add an explicit bound ensuring `getCurrentTick()` cannot materially deviate from TWAP before using it in `getAmountsForLiquidity()`.

## Low/Info Findings

### [L-01] Users can not create their desired vault bacause of `createVault` front-running

**Fixed**

**Description**

`HypoVaultFactory.createVault()` deploys vault clones using `Clones.cloneDeterministic()` with a caller-supplied `salt` that does not incorporate `msg.sender`:

```
// HypoVaultFactory.sol:68
vault = payable(Clones.cloneDeterministic(hypoVaultReference, salt));
```

Because the deterministic address is derived from `keccak256(0xff ++ factory ++ salt ++ initCodeHash)`, and `msg.sender` plays no role in the salt, every caller competes for the same salt namespace. An attacker observing a pending `createVault` transaction in the mempool can extract the salt and front-run the victim with identical salt but attacker-controlled parameters - most critically, `manager`.

### Impact

The attacker deploys a vault at the exact address the victim expected their vault to occupy, initialised with a `manager` the attacker controls. The victim's transaction reverts because the address of the vault is already deployed.

```
// HypoVault.sol:619-625
function manage(
    address target,
    bytes calldata data,
    uint256 value
) external onlyManager returns (bytes memory result) {
    result = target.functionCallWithValue(data, value);
}
```

### Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {HypoVaultFactory} from "src/HypoVaultFactory.sol";
import {HypoVault}        from "src/HypoVault.sol";
import {IVaultAccountant} from "src/interfaces/IVaultAccountant.sol";
import {Clones}           from "lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/proxy/Clones.sol";
import {Errors as OZErrors} from "lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/utils/Errors.sol";

contract SaltFrontrunTest is Test {
    HypoVaultFactory factory;
    address          vaultImpl;

    address victim   = makeAddr("victim");
    address attacker = makeAddr("attacker");
    address token    = makeAddr("token");
    IVaultAccountant accountant = IVaultAccountant(makeAddr("accountant"));

    bytes32 constant SALT = bytes32(uint256(0xdeadbeef));

    function setUp() public {
        vaultImpl = address(new HypoVault());
        factory   = new HypoVaultFactory(vaultImpl);
    }

    function test_saltFrontrun() public {
        // 1. Pre-compute the address the victim expects their vault at.
        address expectedVault = Clones.predictDeterministicAddress(factory.hypoVaultReference(), SALT, address(factory));

        // 2. Attacker front-runs with the same salt but their own manager address.
        vm.prank(attacker);
        address deployedVault = factory.createVault(
            token,
            attacker,   // <-- attacker becomes manager
            accountant,
            500,
            "avUSDC",
            "Attacker Vault USDC",
            SALT
        );

        // 3. Victim's transaction now reverts -- address already taken.
        vm.prank(victim);
        vm.expectRevert(abi.encodeWithSelector(OZErrors.FailedDeployment.selector));
```

```
        factory.createVault(
            token,
            victim,
            accountant,
            500,
            "vUSDC",
            "Victim Vault USDC",
            SALT
        );

        // 4. The vault at the expected address is controlled by the attacker.
        assertEq(deployedVault, expectedVault);
        assertEq(HypoVault(payable(expectedVault)).manager(), attacker);
    }
}
```

**Recommended mitigation**

Incorporate `msg.sender` into the salt before passing it to `cloneDeterministic`. This namespaces each caller's salt space so no two callers can ever collide and stop the anouing front running attempts:

```
function createVault(
    address underlyingToken,
    address manager,
    IVaultAccountant accountant,
    uint256 performanceFeeBps,
    string memory symbol,
    string memory name,
    bytes32 salt
) external returns (address payable vault) {
    bytes32 effectiveSalt = keccak256(abi.encode(msg.sender, salt));
    vault = payable(Clones.cloneDeterministic(hypoVaultReference, effectiveSalt));
    // ...
}
```

### [L-02] Performance Fee Bypassed by Cross-Address PnL Netting via Share Transfer

**Acknowledged**

**Description**

`HypoVault` computes the performance fee independently per address at the moment of withdrawal fulfillment:

```
// HypoVault.sol L496-500
uint256 withdrawnBasis = (uint256(pendingWithdrawal.basis) *
    _withdrawalEpochState.sharesFulfilled) / _withdrawalEpochState.sharesWithdrawn;
uint256 performanceFee = (uint256(
    Math.max(0, int256(assetsToWithdraw) - int256(withdrawnBasis))
) * performanceFeeBps) / 10_000;
```

The `Math.max(0, ...)` clamps negative PnL to zero - a losing withdrawal pays no fee, and that loss does not offset a profit recorded elsewhere. An entity that controls two addresses - one with unrealised profit and one with unrealised loss - can consolidate both positions into the losing address before withdrawing, netting the PnL inside a single `max()` call and eliminating the fee that would have been owed on the profit.

The mechanism is the ERC20 `transfer` / `transferFrom` override, which invokes `_transferBasis`. When the profitable address (lower basis per share) transfers its shares to the losing address (higher basis per share), `_transferBasis` adds the sender's actual basis to the recipient:

```
// HypoVault.sol L609-610
```

```
// reset basis to the lowest average between the sender and receiver to ensure performance
    fee is not deflated
userBasis[to] += Math.min(basisToTransfer, (userBasis[to] * amount) / toBalance);
```

The `Math.min` guard is intended to prevent basis inflation (a high-basis address inflating a low-basis recipient). It does not prevent the opposite: when the sender has a *lower* per-share basis than the recipient, `basisToTransfer < (userBasis[to] * amount) / toBalance`, so `Math.min` resolves to `basisToTransfer` - the full sender basis is transferred, and the combined position's average basis rises to absorb the loss.

**Concrete example (20% performance fee):**

- Address A (profit): 100 shares, basis 1,000 USDC, receives 1,500 USDC, PnL +500 USDC - fee owed **100 USDC**
- Address B (loss): 100 shares, basis 2,000 USDC, receives 1,500 USDC, PnL -500 USDC - fee owed 0
- Separate exit total fee: **100 USDC**
- After A -> B transfer, total fee: **0**

After the transfer, Address B holds 200 shares, basis = 3,000 USDC, receives 3,000 USDC, and pays `max(0, 3000 - 3000)* 20% = 0`. The entire performance fee is eliminated.

**Impact**

Any user controlling two addresses - or cooperating with another user - can suppress performance fees proportionally to the size of offsetting positions. The fee model is therefore sybil-sensitive: a single economic actor split across N addresses pays, at best, 1/N of the fee they would owe if consolidated, and at worst zero if loss positions are available to absorb profits.

**Recommended Mitigation**

In `_transferBasis`, when the sender's per-share basis is lower than the recipient's, apply the fee on the profit delta at transfer time rather than deferring it to withdrawal. This crystallises earned fees when shares move between addresses.

### [L-03] Permissionless executeWithdrawal can front-run user opt-out and force redeposit

**Acknowledged**

**Description**

`HypoVault` allows anyone to call `executeWithdrawal(user, epoch)` once an epoch is fulfilled, while `optOutOfRedeposit(epoch)` can only be called by the user. If a pending withdrawal has shouldRedeposit = true, this creates a race condition:

1. Withdrawal is queued with redeposit enabled.
2. Epoch is fulfilled.
3. User tries to opt out `optOutOfRedeposit`.
4. A third party front-runs with `executeWithdrawal(user, epoch)`.

At that point, execution uses the current stored flag and can route proceeds into redeposit before the user's opt-out is applied. This does not steal funds, but it removes user control over withdrawal destination for that epoch and can force an extra round-trip through deposit/withdraw flows.

**Recommended mitigation** Restrict `executeWithdrawal(user, epoch)` to the user (or approved operator), at least when `shouldRedeposit == true`. Alternatively, require explicit user authorization at execution time for redeposit-enabled withdrawals.

### [I-01] Uninitialized `feeWallet` causes partial withdrawal DoS

**Acknowledged**

**Description**

`HypoVault.initialize()` never assigns the `feeWallet` state variable, leaving it at its default value of `address(0)`:

```
// HypoVault.sol:226-245
function initialize(
    address _underlyingToken,
    address _manager,
    IVaultAccountant _accountant,
    uint256 _performanceFeeBps,
    string memory _symbol,
    string memory _name
) external initializer {
    __Ownable_init(_manager);
    if (_performanceFeeBps > 10_000) revert PerformanceFeeTooHigh();
    underlyingToken    = _underlyingToken;
    manager            = _manager;
    accountant         = _accountant;
    performanceFeeBps  = _performanceFeeBps;
    _internalSupply    = 1_000_000;
    symbol             = _symbol;
    name               = _name;
    // feeWallet is never set
}
```

`performanceFeeBps` can be set to any non-zero value at initialisation time. Whenever a user's withdrawal is profitable, `executeWithdrawal()` deducts a fee from the user's proceeds and transfers it to `feeWallet`:

```
// HypoVault.sol:498-525
uint256 performanceFee = (uint256(
            Math.max(0, int256(assetsToWithdraw) - int256(withdrawnBasis))
        ) * performanceFeeBps) / 10_000;

// ...

if (performanceFee > 0) {
    assetsToWithdraw -= performanceFee;
        // user loses the fee
    SafeTransferLib.safeTransfer(underlyingToken, feeWallet, uint256(performanceFee));
        // transfer reverts because feeWallet == 0
}
```

However, as in or case feeWallet is address(0) initially the transfer fails and the user can not executeWithdrawal timely.

**Impact** Two distinct failure modes arise depending on the underlying token:

- Reverts on `transfer(to = address(0))` - standard OpenZeppelin ERC-20. Every call to `executeWithdrawal` for a profitable position reverts. Withdrawal processing is **completely blocked** for all such users until the owner manually calls `setFeeWallet`.
- Allows `transfer(to = address(0))` - effectively burns the token. Performance fees are silently and permanently destroyed. The protocol collects zero revenue and the tokens are unrecoverable.

In both cases, since there is no enforced ordering between vault deployment and fee wallet configuration, and vaults are usable immediately after `createVault` returns, the vulnerable window begins at block 0 of the vault's life. Any vault deployed with `performanceFeeBps > 0` and used before `setFeeWallet` is called is affected.

**Proof of Concept**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import {HypoVaultFactory} from "src/HypoVaultFactory.sol";
import {HypoVault}        from "src/HypoVault.sol";
import {IVaultAccountant} from "src/interfaces/IVaultAccountant.sol";
import {ERC20Mock}        from "lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/mocks/token/ERC20Mock.sol";

contract MockAccountant is IVaultAccountant {
    uint256 public nav;
    function setNAV(uint256 _nav) external { nav = _nav; }
    function computeNAV(address, address, bytes memory) external view returns (uint256) {
        return nav;
    }
}

contract UninitializedFeeWalletTest is Test {
    HypoVaultFactory factory;
    HypoVault        vault;
    ERC20Mock        token;
    MockAccountant   accountant;

    address manager = makeAddr("manager");
    address user    = makeAddr("user");

    function setUp() public {
        token      = new ERC20Mock();
        accountant = new MockAccountant();

        address vaultImpl = address(new HypoVault());
        factory = new HypoVaultFactory(vaultImpl);

        // Deploy vault with 10% performance fee -- feeWallet is NOT set.
        address payable vaultAddr = factory.createVault(
            address(token),
            manager,
            IVaultAccountant(address(accountant)),
            1_000,   // 10% performance fee
            "vTKN",
            "Vault Token",
            bytes32(uint256(1))
        );
        vault = HypoVault(vaultAddr);

        // Confirm feeWallet is address(0).
        assertEq(vault.feeWallet(), address(0));
    }

    function test_withdrawalDosWhenFeeWalletUnset() public {
        // 1. User deposits 1000 tokens.
        token.mint(user, 1_000e18);
        vm.startPrank(user);
        token.approve(address(vault), type(uint256).max);
        vault.requestDeposit(1_000e18);
        vm.stopPrank();

        // 2. Manager fulfills the deposit at a 1:1 NAV.
        accountant.setNAV(1_000e18);
        vm.prank(manager);
        vault.fulfillDeposits(1_000e18, "");
```

```
62
63          // 3. User claims shares.
64          vault.executeDeposit(user, 0);
65
66          // 4. User requests a withdrawal.
67          uint128 shares = uint128(vault.balanceOf(user));
68          vm.prank(user);
69          vault.requestWithdrawal(shares);
70
71          // 5. NAV grows -- position is now profitable; performance fee will be > 0.
72          accountant.setNAV(2_000e18);
73
74          // 6. Manager fulfills the withdrawal.
75          token.mint(address(vault), 2_000e18);
76          vm.prank(manager);
77          vault.fulfillWithdrawals(shares, type(uint256).max, "");
78
79          // 7. executeWithdrawal reverts because SafeTransferLib sends fee to address(0)
80          //    and the underlying ERC-20 reverts on transfer to the zero address.
81          vm.expectRevert();
82          vault.executeWithdrawal(user, 0);
83
84          // User's profitable withdrawal is permanently stuck.
85      }
86  }
```

**Recommended mitigation**

Add a `_feeWallet` parameter to `initialize()` and assign it immediately:

In HypoVault:

```
1   function initialize(
2       address _underlyingToken,
3       address _manager,
4       IVaultAccountant _accountant,
5       uint256 _performanceFeeBps,
6       address _feeWallet,          // added
7       string memory _symbol,
8       string memory _name
9   ) external initializer {
10      __Ownable_init(_manager);
11
12      if (_performanceFeeBps > 10_000) revert PerformanceFeeTooHigh();
13      if (_performanceFeeBps > 0 && _feeWallet == address(0)) revert InvalidFeeWallet(); //
            added
14
15      underlyingToken   = _underlyingToken;
16      manager           = _manager;
17      accountant        = _accountant;
18      performanceFeeBps = _performanceFeeBps;
19      feeWallet         = _feeWallet;          // added
20      _internalSupply   = 1_000_000;
21      symbol            = _symbol;
22      name              = _name;
23  }
```

In HypoVaultFactory:

```
1   /// @notice Creates a new HypoVault instance.
2       /// @param underlyingToken The token used to denominate deposits and withdrawals
3       /// @param manager The account authorized to execute deposits, withdrawals, and make
            arbitrary function calls from the vault
4       /// @param accountant The contract that reports the net asset value of the vault
```

```
    /// @param performanceFeeBps The performance fee, in basis points, taken on each
        profitable withdrawal
    /// @param symbol The symbol of the share token
    /// @param name The name of the share token
    /// @return vault The address of the newly created vault
    function createVault(
        address underlyingToken,
        address manager,
        IVaultAccountant accountant,
        uint256 performanceFeeBps,
        address feeWallet,
        string memory symbol,
        string memory name,
        bytes32 salt
    ) external returns (address payable vault) {
        // NOTE shouldn't msg.sender be added to the salt.
        vault = payable(Clones.cloneDeterministic(hypoVaultReference, salt));

        // NOTE feeWallet is not set in the vault, should it be?
        HypoVault(vault).initialize(
            underlyingToken,
            manager,
            accountant,
            performanceFeeBps,
            feeWallet,
            symbol,
            name
        );
```

Propagate `_feeWallet` through `HypoVaultFactory.createVault()` accordingly and add the same non-zero guard to `setFeeWallet()` to prevent the zero-address state from being re-introduced after deployment.