BaseLedger Contract Accounting Vulnerability Report

BaseLedger._updateAccounting Function Fails to Update User Accounting State During Zero-Fee Outflow Operations

MEDIUM SEVERITY

Summary

When feePercent is set to 0, the BaseLedger contract skips the entire outflow processing logic, leaving user share balances and cost basis unchanged despite actual asset withdrawals.

Finding Description

The BaseLedger._updateAccounting function contains a critical flaw in its outflow processing logic. When processing outflow operations, the function contains a conditional check that only processes accounting updates if config.feePercent != 0:

```
} else {
    // Only process outflow if feePercent is not set to 0
    if (config.feePercent != 0) {
        uint256 amountAssets = _getOutflowProcessVolume(
            amountSharesOrAssets,
            usedShares,
            pps,
            IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
        );

        feeAmount = _processOutflow(user, yieldSource, amountAssets, usedShares
        // ... accounting updates occur here ...
```

```
            return feeAmount;
        } else {
            emit AccountingOutflowSkipped(user, yieldSource, yieldSourceOracleId, a
            return 0;
        }
    }
```

When feePercent is 0, the function:

1. Emits AccountingOutflowSkipped event
2. Returns 0 fee amount
3. Fails to call _processOutflow which contains _calculateCostBasis
4. Leaves usersAccumulatorShares and usersAccumulatorCostBasis unchanged

The _calculateCostBasis function is only called within _processOutflow, meaning zero-fee withdrawals bypass all accounting updates entirely.

Impact Explanation

- **Inflated Balances:** User's share balance and cost basis remain inflated after withdrawals
- **Accounting Mismatch:** The contract's accounting does not reflect the user's actual position
- **Persistent Corruption:** The corrupted state remains even after fee percentages are restored to non-zero values
- **Protocol Integrity:** Future calculations based on corrupted state will produce incorrect results
- **Financial Risk:** Users may be able to withdraw more than their fair share due to inflated accounting

Proof of Concept

A test contract in the test folder. Run with the command:

> ```
> forge test --mc CostBasisCorruptionTest -vvvv
> ```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

import "forge-std/Test.sol";
import "../src/core/accounting/SuperLedger.sol";
import "../src/core/accounting/SuperLedgerConfiguration.sol";
import "../src/core/accounting/oracles/ERC4626YieldSourceOracle.sol";
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import { Mock4626Vault } from "test/mocks/Mock4626Vault.sol";
import { MockERC20 } from "test/mocks/MockERC20.sol";

contract CostBasisCorruptionTest is Test {
    SuperLedger ledger;
    SuperLedgerConfiguration config;
    ERC4626YieldSourceOracle oracle;
    Mock4626Vault vault;
    MockERC20 asset;

    address user = address(0x1);
    address executor = address(0x3);
    bytes4 yieldSourceOracleId = bytes4(0x12345678);
    address yieldSource;

    function setUp() public {
        // Deploy mock contracts first
        asset = new MockERC20("Test Asset", "TST", 18);
        vault = new Mock4626Vault(address(asset), "Test Vault", "TVLT");
        yieldSource = address(vault);

        // Deploy contracts
        address[] memory allowedExecutors = new address[](1);
        allowedExecutors[0] = executor;

        config = new SuperLedgerConfiguration();
        ledger = new SuperLedger(address(config), allowedExecutors);
        oracle = new ERC4626YieldSourceOracle();

        ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs
            new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);

        configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: yieldSourceOracleId,
            yieldSourceOracle: address(oracle),
```

```
            feePercent: 1000, // 10% fee
            feeRecipient: address(0x999),
            ledger: address(ledger)
        });

        config.setYieldSourceOracles(configs);
    }

    function testCostBasisCorruptionWithZeroFee() public {
        vm.startPrank(executor);

        // Step 1: User deposits 100 shares (inflow)
        ledger.updateAccounting(
            user,
            yieldSource,
            yieldSourceOracleId,
            true, // isInflow
            100e18, // amountSharesOrAssets
            0 // usedShares (not used for inflow)
        );

        // Verify initial state
        uint256 initialShares = ledger.usersAccumulatorShares(user, yieldSource
        uint256 initialCostBasis = ledger.usersAccumulatorCostBasis(user, yield

        assertEq(initialShares, 100e18);
        assertEq(initialCostBasis, 100e18); // 1:1 ratio

        // Step 2: Change fee to 0
        ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory newConfi
            new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);

        newConfigs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: yieldSourceOracleId,
            yieldSourceOracle: address(oracle),
            feePercent: 0, // Zero fee
            feeRecipient: address(0x999),
            ledger: address(ledger)
        });

        vm.stopPrank();
        config.proposeYieldSourceOracleConfig(newConfigs);

        // Fast forward past proposal expiration time (1 week)
        vm.warp(block.timestamp + 8 days);

        bytes4[] memory idsToAccept = new bytes4[](1);
        idsToAccept[0] = yieldSourceOracleId;
        config.acceptYieldSourceOracleConfigProposal(idsToAccept);

        vm.startPrank(executor);
```

```
// Step 3: User withdraws 50 shares with zero fee (outflow)
uint256 feeAmount = ledger.updateAccounting(
    user,
    yieldSource,
    yieldSourceOracleId,
    false, // isOutflow
    50e18, // amountSharesOrAssets
    50e18 // usedShares
);

// Verify fee is 0 (as expected)
assertEq(feeAmount, 0);

// Step 4: Check accounting corruption - shares and cost basis should N
uint256 sharesAfterWithdrawal = ledger.usersAccumulatorShares(user, yie
uint256 costBasisAfterWithdrawal = ledger.usersAccumulatorCostBasis(use

// BUG: These should be reduced by 50e18 each, but they remain unchange
assertEq(sharesAfterWithdrawal, 100e18, "Shares should be reduced but w
assertEq(costBasisAfterWithdrawal, 100e18, "Cost basis should be reduce

// Step 5: Change fee back to non-zero to demonstrate persistent corrup
ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory restoreC
    new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);

restoreConfigs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigAr
    yieldSourceOracleId: yieldSourceOracleId,
    yieldSourceOracle: address(oracle),
    feePercent: 1000, // 10% fee restored
    feeRecipient: address(0x999),
    ledger: address(ledger)
});

vm.stopPrank();
config.proposeYieldSourceOracleConfig(restoreConfigs);
vm.warp(block.timestamp + 8 days);
config.acceptYieldSourceOracleConfigProposal(idsToAccept);

vm.startPrank(executor);

// Calculate expected cost basis for remaining 50 shares
uint256 expectedCostBasis = ledger.calculateCostBasisView(user, yieldSo

// This will show 50e18 instead of 25e18 due to corruption
assertEq(expectedCostBasis, 50e18, "Cost basis calculation affected by

vm.stopPrank();
```

```
        }
    }
```

Recommendation

Solution: Always Update Accounting for Outflow Operations

Modify the _updateAccounting function to always update user accounting for outflow operations, regardless of fee percentage:

```
} else {
    uint256 amountAssets = _getOutflowProcessVolume(
        amountSharesOrAssets,
        usedShares,
        pps,
        IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
    );

    if (config.feePercent != 0) {
        feeAmount = _processOutflow(user, yieldSource, amountAssets, usedS
        emit AccountingOutflow(user, config.yieldSourceOracle, yieldSource
    } else {
        // Update accounting even with zero fees
        _calculateCostBasis(user, yieldSource, usedShares);
        emit AccountingOutflow(user, config.yieldSourceOracle, yieldSource
    }
    return feeAmount;
}
```

Key Changes:

- Always call _calculateCostBasis for outflow operations
- Properly update usersAccumulatorShares and usersAccumulatorCostBasis
- Emit appropriate events for all outflow operations
- Maintain accounting integrity regardless of fee configuration

Additional Considerations

This vulnerability highlights the importance of:

- Separating fee calculation from core accounting logic
- Comprehensive testing of edge cases, including zero-value configurations
- Clear documentation of intended behavior for all configuration states
- Regular audits of state-changing functions to ensure data consistency