// Security Assessment

03.18.2026 - 03.23.2026

Earn v2: Hurdle Rate

Blueprint Finance

HALBORN

Earn v2: Hurdle Rate - Blueprint Finance

Prepared by: HALBORN

Last Updated 04/22/2026

Date of Engagement: March 18th, 2026 - March 23rd, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	0	2	5

TABLE OF CONTENTS

1. INTRODUCTION

`Blueprint Finance` engaged `Halborn` to perform a security assessment of their smart contracts from March 18th, 2026 to March 23rd, 2026. The assessment scope was limited to the smart contracts provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

The `earn-v2-core` vault system is an ERC-4626 compliant upgradeable yield vault where users deposit assets that are allocated across pluggable strategies to generate yield. This review covers the addition of a hurdle rate performance fee mechanism, where performance fees are charged only on yield exceeding a configurable benchmark rate provided by one of three oracle implementations: an APR-based continuous compounding oracle, an APY-based compound growth oracle, or an offchain oracle pushed by a backend keeper for async vaults with discrete NAV updates. The review also covers the FeeAccountant, a UUPS-upgradeable peripheral contract that receives minted fee shares and distributes them to whitelisted recipients via a backend-operated distribution call, and supporting changes to the vault's yield accrual library, storage layout, fee constants, and interface definitions.

2. ASSESSMENT SUMMARY

`Halborn` was allocated 3 days for this engagement and assigned 1 full-time security engineer to conduct a comprehensive review of the smart contracts within scope. The engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the smart contracts.
- Verify that the smart contract functionality operates as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were partially addressed by the `Blueprint Finance` team. The main recommendations were:

- `Deduct the strategy's allocated balance from cachedTotalAssets before deleting strategy data during removal.`
- `Wrap the oracle's external calls in a try-catch so a reverting oracle does not block vault deposits and withdrawals.`
- `Use circulating supply instead of total supply when computing the hurdle threshold to exclude vault-held queued shares.`
- `Add a floor check in yield accrual to cap negative yield at the current cachedTotalAssets value and prevent underflow.`
- `Record a lastUpdated timestamp on each setRate call and enforce a configurable maximum staleness window in getHurdleRate.`

3. TEST APPROACH & METHODOLOGY

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

The following phases and associated tools were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Solidity variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts.
- Fork testing against main networks.
- Static security analysis of scoped contracts, and imported functions (`Slither`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: earn-v2-core

(b) Assessed Commit ID: 93e7e4c

(c) Items in scope:

- src/implementation/ConcreteStandardVaultImpl.sol
- src/interface/IConcreteStandardVaultImpl.sol
- src/interface/IHurdleRateOracle.sol
- src/lib/storage/ConcreteStandardVaultImplStorageLib.sol
- src/lib/Constants.sol
- src/lib/YieldAccrualLib.sol
- src/oracle/APRRateHurdleOracle.sol
- src/oracle/APYRateHurdleOracle.sol
- src/oracle/OffchainHurdleRateOracle.sol
- src/periphery/auxiliary/FeeAccountant.sol

Out-of-Scope: Third party dependencies and economic attacks. Furthermore, this assessment was limited to the differential changes between the earn-2.5 baseline and the dev/earn-2.2 branch at commit 93e7e4c. Components outside of the reviewed diff, pre-existing logic not modified by the changes, system-wide interactions not directly surfaced in the diff context were explicitly out of scope.

FILE

(a) Submitted File: earn-v2-audit-scope.zip

REMEDIATION COMMIT ID:

- 9e725ef
- 85b5105

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	2

INFORMATIONAL
5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
OFFCHAIN HURDLE RATE ORACLE HAS NO STALENESS CHECK	LOW	SOLVED - 03/26/2026
REVERTING ORACLE CREATES SELF-LOCKING VAULT DENIAL OF SERVICE	LOW	RISK ACCEPTED - 03/31/2026
MISSING HIGH-WATER MARK ENABLES DOUBLE PERFORMANCE FEE CHARGING ON RECOUPED LOSSES	INFORMATIONAL	ACKNOWLEDGED - 03/26/2026
DUPLICATE WHENNOTPAUSED MODIFIER ON ALLOCATE FUNCTION	INFORMATIONAL	SOLVED - 03/23/2026
REMOVING HALTED STRATEGY WITH NON-ZERO ALLOCATION CREATES PHANTOM CACHED TOTAL ASSETS	INFORMATIONAL	ACKNOWLEDGED - 03/31/2026
MINIMUM TRANSACTION AMOUNTS NOT REFLECTED IN MAXDEPOSIT AND MAXWITHDRAW VIEW FUNCTIONS	INFORMATIONAL	SOLVED - 03/26/2026
DEFAULT_ADMIN_ROLE RENOUNCEMENT PERMANENTLY BRICKS FEEACCOUNTANT	INFORMATIONAL	ACKNOWLEDGED - 03/26/2026

7. FINDINGS & TECH DETAILS

7.1 OFFCHAIN HURDLE RATE ORACLE HAS NO STALENESS CHECK

// LOW

Description

The `OffchainHurdleRateOracle` stores no `lastUpdated` timestamp and enforces no maximum staleness window. The `getHurdleRate()` function returns whatever value was last set via `setRate()`, regardless of how long ago the update occurred.

The `OffchainHurdleRateOracle` is designed for use cases where an off-chain backend periodically pushes updated hurdle rates that reflect the desired benchmark (for example, a compounded growth rate computed off-chain). The `getHurdleRate()` function simply returns the stored rate value with no check on when it was last written. If the off-chain backend goes offline, encounters an error, or the oracle owner loses access to the update key, the rate remains frozen at its last value indefinitely. For asynchronous vaults where the specification requires NAV updates and yield accrual to happen in the same batched transaction, a stale hurdle rate means the fee calculation compares current yield against an outdated benchmark. If the intended hurdle should have grown over time (as with APR or APY-based benchmarks), a stale rate is lower than the intended current value, exposing more yield to performance fees than the depositors should bear.

Code Location

`OffchainHurdleRateOracle.sol`, `getHurdleRate()` with no staleness check:

Copy Code

```
/// @inheritdoc IHurdleRateOracle
function getHurdleRate() external view override returns (uint256) {
    return rate;
    // Returns stored value regardless of age
    // No lastUpdated timestamp tracked
    // No maxStaleness enforced
}
```

`OffchainHurdleRateOracle.sol`, `setRate()` does not record update timestamp:

Copy Code

```
function setRate(uint256 newRate_) external onlyOwner {
    if (newRate_ == 0) revert ZeroRate();
    if (newRate_ > MAX_RATE) revert RateTooHigh(newRate_, MAX_RATE);
    uint256 oldRate = rate;
    rate = newRate_;
    // No lastUpdated = block.timestamp
    emit RateUpdated(oldRate, newRate_);
}
```

`OffchainHurdleRateOracle.sol`, storage has no timestamp or staleness fields:

```
contract OffchainHurdleRateOracle is IHurdleRateOracle, Ownable {
    uint256 private constant _PRECISION = 1e18;

    uint256 public rate;
    // No lastUpdated field
    // No maxStaleness field

}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:L (3.1)

Recommendation

Consider adding a staleness check to `getHurdleRate()` and recording the update timestamp in `setRate()`. This ensures the vault reverts or degrades gracefully when the oracle has not been updated within an acceptable window, rather than silently using an outdated rate.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this finding in the specified commit by adding a `lastUpdateTimestamp` field that is set on construction and updated on every `setRate()` call, a configurable `maxStaleness` threshold defaulting to 100 years, and a staleness check in `getHurdleRate()` that reverts with `StaleRate` when the rate age exceeds the configured maximum.

Client Note:
We have decided to add an optional staleness check, configured on case by case basis depending on each deployment requirements. For a typical deployment this check is, however, redundant since both hurdle update and NAV update will happen in the same transaction (both or nothing).

Remediation Hash

9e725ef892d5ed6d299d3bd7d4950a04ca3764dc

7.2 REVERTING ORACLE CREATES SELF-LOCKING VAULT DENIAL OF SERVICE

// LOW

Description

If a configured hurdle rate oracle reverts on `getHurdleRate()` or `precision()`, all vault operations guarded by the `withYieldAccrual` modifier permanently revert, including the `setHurdleRateOracle()` function that would be used to replace the broken oracle. The circular dependency between oracle calls and the yield accrual modifier locks the vault into a state where deposits, withdrawals, and administrative recovery are all blocked.

The `withYieldAccrual` modifier calls `accrueYield()` before executing the body of any decorated function. During yield accrual, fetchHurdleRate() makes unprotected external calls to oracle.getHurdleRate() and oracle.precision(). If the oracle contract reverts for any reason, the entire transaction reverts before reaching the function body. Because `deposit()`, `mint()`, `withdraw()`, `redeem()`, and `allocate()` all carry the `withYieldAccrual` modifier, every user-facing operation is blocked.

Code Location

`ConcreteStandardVaultImpl.sol`, `setHurdleRateOracle()` decorated with `withYieldAccrual`:

Copy Code

```
function setHurdleRateOracle(IHurdleRateOracle oracle)
    external
    onlyRole(RolesLib.VAULT_MANAGER)
    withYieldAccrual  // <-- calls _accrueYield(), which calls the broken oracle
{
    SVLib.fetch().hurdleRateOracle = oracle;
    emit HurdleRateOracleUpdated(address(oracle));
}
```

`ConcreteStandardVaultImpl.sol`, `withYieldAccrual` modifier:

Copy Code

```
modifier withYieldAccrual() {
    _accrueYield(); // <-- reverts if oracle reverts
    _;
}
```

`YieldAccrualLib.sol`, `_fetchHurdleRate()` with unprotected external calls:

Copy Code

```
function _fetchHurdleRate(IHurdleRateOracle oracle)
    private
    view
    returns (uint256 hurdleRate, uint256 oraclePrecision)
{
    if (address(oracle) == address(0)) return (0, 0);
    hurdleRate = oracle.getHurdleRate();   // <-- no try/catch
    oraclePrecision = oracle.precision();  // <-- no try/catch
}
```

`ConcreteStandardVaultImpl.sol`, user-facing functions also blocked:

```
function deposit(uint256 assets, address receiver)
    public virtual override whenNotPaused nonReentrant
    withYieldAccrual  // <-- same deadlock path
    returns (uint256) { ... }

function withdraw(uint256 assets, address receiver, address owner)
    public virtual override whenNotPaused nonReentrant
    withYieldAccrual  // <-- same deadlock path
    returns (uint256) { ... }
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:H/I:N/D:M/Y:N (2.9)

Recommendation

Consider removing the `withYieldAccrual` modifier from `setHurdleRateOracle()`. The oracle replacement does not depend on prior yield accrual being settled, as the next user operation will accrue yield using the newly set oracle. Additionally, wrapping the external calls in `_fetchHurdleRate()` with a try/catch would allow the vault to degrade gracefully when the oracle is unresponsive, falling back to (0, 0) and charging the full performance fee as if no hurdle were configured.

Remediation Comment

RISK ACCEPTED: The **Blueprint Finance team** made a business decision to accept the risk of this finding and not alter the contracts.

Client Note:
We acknowledge that this finding is technically correct, but we accept the risk as this behavior is by design and we consider it low severity. The `accrueYield` call in `setHurdleRateOracle()` is intentional, as it ensures any outstanding yield is accounted for and the performance fee is applied before the oracle is changed, avoiding retroactive fee accounting issues. Current hurdle oracle implementations cannot revert in a non-recoverable way, and we prefer to keep the vault interface lean rather than add unnecessary emergency functions at this stage. If we ever introduce an oracle implementation that could become permanently broken, we can address that at the implementation level by adding an emergency removal function in a future upgrade.

7.3 MISSING HIGH-WATER MARK ENABLES DOUBLE PERFORMANCE FEE CHARGING ON RECOUPED LOSSES

// INFORMATIONAL

Description

The storage struct declares a `performanceFeeHighWaterMark` field (uint128), but it is never read or written anywhere in the codebase. Without a high-water mark, performance fees are charged on every positive net yield accrual event regardless of whether the vault has recovered from prior losses.

Depositors are charged performance fees on yield that merely recovers previous losses rather than creating new value. The over-extraction scales with market volatility and accrual frequency. In a scenario with monthly accruals on a strategy that alternates between +10% and -10%, depositors pay performance fees on every positive month while receiving no credit for negative months.

Code Location

`ConcreteStandardVaultImplStorageLib.sol`, unused high-water mark field:

Copy Code

```
struct ConcreteStandardVaultImplStorage {
    // ...
    /// @dev annual performance fee rate in basis points
    uint16 performanceFee;
    /// @dev high water mark
    uint128 performanceFeeHighWaterMark;
    // Declared but NEVER read or written anywhere in the codebase
    // ...
}
```

`YieldAccrualLib.sol`, `accruePerformanceFee()` charges fees without HWM check:

Copy Code

```
function accruePerformanceFee(uint256 totalAssetsAmount, uint256 positiveYield, uint256 loss) public
    SVLib.ConcreteStandardVaultImplStorage storage $ = SVLib.fetch();

    if ($.performanceFee == 0 || (loss >= positiveYield)) return;
    // Only checks if current losses exceed current gains
    // No check against historical peak (high-water mark)

    uint256 totalSupply_ = ERC20_OZ_5_2_0_Lib.totalSupply();

    (uint256 hurdleRate, uint256 oraclePrecision) = _fetchHurdleRate($.hurdleRateOracle);

    (uint256 performanceFeeShares, uint256 feeAmount) = _previewPerformanceFee(
        totalAssetsAmount, positiveYield, loss, totalSupply_, $.performanceFee, hurdleRate, oraclePre
    );
    // Fees computed on any net positive yield, even if vault is below its all-time high

    if (performanceFeeShares == 0) return;

    address performanceFeeRecipient = $.performanceFeeRecipient;
    if (performanceFeeRecipient != address(0)) {
        ERC20_OZ_5_2_0_Lib._mint(performanceFeeRecipient, performanceFeeShares);
        // Fee shares minted without comparing totalAssetsAmount to a stored peak
        // ...
    }
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

If the high-water mark is intentionally omitted as a design decision, consider removing the dead `performanceFeeHighWaterMark` storage field to avoid confusion and documenting the design choice prominently in the contract's NatSpec. If the high-water mark should be active, consider implementing HWM tracking in `accruePerformanceFee()` so that fees are only charged when total assets exceed the previous peak.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** made a business decision to acknowledge this finding and not alter the contracts.

Client Note:
We consciously chose this design after carefully considering the alternatives and prevailing industry practices. We view it as a deliberate design choice, our fees are calculated against periodically settled net yield and do not cause the share value to decline. This approach is consistent with that of major DeFi protocols, which handle this situation in a similar manner. Furthermore, when combined with a hurdle rate, the hurdle rate guarantees a minimum level of returns that is exempt from future performance fees. In this way, the hurdle rate effectively functions as a soft high-water mark.

7.4 DUPLICATE WHENNOTPAUSED MODIFIER ON ALLOCATE FUNCTION

// INFORMATIONAL

Description

The `allocate()` function applies the whenNotPaused modifier twice in its declaration. The duplicate has no functional impact because the check is idempotent, but it wastes gas on a redundant paused() storage read and indicates a code review gap.

Code Location

`ConcreteStandardVaultImpl.sol`, `allocate()` with duplicate modifier:

Copy Code

```
function allocate(bytes calldata data)
    public
    virtual
    whenNotPaused       // First occurrence
    nonReentrant
    onlyRole(RolesLib.ALLOCATOR)
    withYieldAccrual
    whenNotPaused       // Duplicate -- redundant
{
    // delegatecall allocate module
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

Consider removing the duplicate `whenNotPaused` modifier.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this finding in the specified commit by removing the duplicate `whenNotPaused` modifier from the `allocate()` function, retaining the single existing instance that correctly enforces the pause check.

Client Note:
This was independently identified by our automations and fixed in a separate pull request pending internal review.

Remediation Hash

85b51059844fdfebb9d404f0c76d94684c32652b

7.5 REMOVING HALTED STRATEGY WITH NON-ZERO ALLOCATION CREATES PHANTOM CACHED TOTAL ASSETS

// INFORMATIONAL

Description

The `removeStrategy()` function permits removal of a halted strategy even when it holds a non-zero allocation, but does not reduce `cachedTotalAssets` by the removed allocation amount. The resulting phantom balance permanently inflates the vault's share price, and the vault eventually becomes insolvent as withdrawers drain real assets against an overstated accounting total.

In `StateSetterLib.sol`, `removeStrategy()` enforces that a strategy must either have zero allocation or be in a Halted status before it can be removed. The Halted status check bypasses the zero-allocation requirement, which means a strategy carrying a significant allocation can be deleted from storage via delete `$.strategyData[strategy]`. This delete operation zeroes the strategy's allocated field and removes the strategy from the enumerable set. However, `cachedTotalAssets` is not adjusted to reflect the loss of the deleted allocation. From that point forward, the vault's internal accounting references assets that no longer exist in any tracked strategy, and the share price is computed against a total that exceeds the vault's actual holdings.

The phantom balance causes direct, irreversible fund loss. The vault's share price is permanently inflated by the amount of the stranded allocation, meaning every subsequent withdrawal pays out more than the vault can sustain. Last withdrawers will be unable to redeem their shares. The inflated cachedTotalAssets also feeds into management fee calculations, generating fees on non-existent assets. New depositors who enter after the removal receive fewer shares than they should, because the price-per-share is artificially high.

Code Location

`StateSetterLib.sol`, `removeStrategy()` function:

Copy Code

```
function removeStrategy(address strategy) external {
    SVLib.ConcreteStandardVaultImplStorage storage $ = SVLib.fetch();

    IConcreteStandardVaultImpl.StrategyData memory strategyDataCached = $.strategyData[strategy];

    require(
        (strategyDataCached.allocated == 0 && _strategyNotInDeallocationOrder(strategy))
            || strategyDataCached.status == IConcreteStandardVaultImpl.StrategyStatus.Halted,
            // Halted status bypasses the allocated == 0 check
        IConcreteStandardVaultImpl.StrategyHasAllocation()
    );
    require($.strategies.remove(strategy), IConcreteStandardVaultImpl.StrategyDoesNotExist());

    delete $.strategyData[strategy];
    // Zeroes allocated field, but cachedTotalAssets is never reduced
    _removeStrategyFromDeallocationOrder(strategy);

    emit IConcreteStandardVaultImpl.StrategyRemoved(strategy);
}
```

Proof of Concept

Copy Code

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import {ConcreteStandardVaultImplBaseSetup} from "../../common/ConcreteStandardVaultImplBaseSetup.t.s
import {AddStrategyWithDeallocationOrder} from "../../common/AddStrategyWithDeallocationOrder.sol";
import {IConcreteStandardVaultImpl} from "../../../src/interface/IConcreteStandardVaultImpl.sol";
import {IAllocateModule} from "../../../src/interface/IAllocateModule.sol";
import {ERC4626StrategyMock} from "../../mock/ERC4626StrategyMock.sol";
import {ConcreteV2RolesLib as RolesLib} from "../../../src/lib/Roles.sol";

contract C01_PhantomCachedTotalAssets is ConcreteStandardVaultImplBaseSetup, AddStrategyWithDeallocat
    ERC4626StrategyMock public strategy;

    address public alice;
    address public bob;

    uint256 constant DEPOSIT_AMOUNT = 500_000e18;

    function setUp() public override {
        super.setUp();

        alice = makeAddr("alice");
        bob = makeAddr("bob");

        // Deploy and register strategy
        strategy = new ERC4626StrategyMock(address(asset), address(concreteStandardVault));
        addStrategyWithDeallocationOrder(
            address(strategy),
            address(concreteStandardVault),
            allocator,
            strategyOperator
        );

        // Alice and Bob each deposit 500k tokens
        asset.mint(alice, DEPOSIT_AMOUNT);
        asset.mint(bob, DEPOSIT_AMOUNT);

        vm.startPrank(alice);
        asset.approve(address(concreteStandardVault), DEPOSIT_AMOUNT);
        concreteStandardVault.deposit(DEPOSIT_AMOUNT, alice);
        vm.stopPrank();

        vm.startPrank(bob);
        asset.approve(address(concreteStandardVault), DEPOSIT_AMOUNT);
        concreteStandardVault.deposit(DEPOSIT_AMOUNT, bob);
        vm.stopPrank();

        // Allocate 100% to strategy
        uint256 totalToAllocate = concreteStandardVault.cachedTotalAssets();
        IAllocateModule.AllocateParams[] memory params = new IAllocateModule.AllocateParams[](1);
        params[0] = IAllocateModule.AllocateParams({
            isDeposit: true,
            strategy: address(strategy),
            extraData: abi.encode(totalToAllocate)
        });

        vm.prank(allocator);
        concreteStandardVault.allocate(abi.encode(params));
    }

    function test_C01_phantomCachedTotalAssets_vaultInsolvency() public {
        uint256 totalDeposited = DEPOSIT_AMOUNT * 2; // 1,000,000e18

        // --- Snapshot before ---
        uint256 cachedBefore = concreteStandardVault.cachedTotalAssets();
        IConcreteStandardVaultImpl.StrategyData memory dataBefore =
            concreteStandardVault.getStrategyData(address(strategy));

        assertEq(cachedBefore, totalDeposited, "cachedTotalAssets should equal total deposits");
        assertEq(dataBefore.allocated, totalDeposited, "strategy should hold full allocation");

        // --- Step 1: Halt the strategy (natural incident response) ---
```

```
        vm.prank(strategyOperator);
        concreteStandardVault.toggleStrategyStatus(address(strategy));

        IConcreteStandardVaultImpl.StrategyData memory dataAfterHalt =
            concreteStandardVault.getStrategyData(address(strategy));
        assertEq(
            uint8(dataAfterHalt.status),
            uint8(IConcreteStandardVaultImpl.StrategyStatus.Halted),
            "strategy should be Halted"
        );

        // --- Step 2: Remove the halted strategy (bypasses allocated == 0 check) ---
        vm.prank(strategyOperator);
        concreteStandardVault.removeStrategy(address(strategy));

        // --- Verify the bug: cachedTotalAssets is NOT reduced ---
        uint256 cachedAfterRemoval = concreteStandardVault.cachedTotalAssets();
        assertEq(
            cachedAfterRemoval,
            totalDeposited,
            "BUG: cachedTotalAssets still includes the removed allocation"
        );

        // Strategy data is deleted, but the vault thinks it still has 1,000,000e18
        // Real assets in vault's possession: 0 (all were in the strategy)
        uint256 vaultBalance = asset.balanceOf(address(concreteStandardVault));
        assertEq(vaultBalance, 0, "vault holds zero real tokens");

        // --- Step 3: Demonstrate insolvency ---
        // Alice tries to withdraw her 500k. Share price is inflated because
        // cachedTotalAssets says 1,000,000 but vault has 0.
        uint256 aliceShares = concreteStandardVault.balanceOf(alice);
        uint256 aliceMaxWithdraw = concreteStandardVault.maxWithdraw(alice);

        // maxWithdraw reports a non-zero amount based on inflated cachedTotalAssets
        // but the vault cannot fulfill it
        vm.startPrank(alice);
        if (aliceMaxWithdraw > 0) {
            vm.expectRevert(); // Insufficient real balance
            concreteStandardVault.withdraw(aliceMaxWithdraw, alice, alice);
        }
        vm.stopPrank();

        // Bob also cannot withdraw
        uint256 bobShares = concreteStandardVault.balanceOf(bob);
        uint256 bobMaxWithdraw = concreteStandardVault.maxWithdraw(bob);

        vm.startPrank(bob);
        if (bobMaxWithdraw > 0) {
            vm.expectRevert();
            concreteStandardVault.withdraw(bobMaxWithdraw, bob, bob);
        }
        vm.stopPrank();

        // --- Final assertion: phantom balance proven ---
        // The vault reports assets it cannot back
        uint256 reportedTotal = cachedAfterRemoval;
        uint256 actualTotal = asset.balanceOf(address(concreteStandardVault));
        uint256 phantomAmount = reportedTotal - actualTotal;

        assertEq(phantomAmount, totalDeposited, "phantom balance equals the full removed allocation")

        emit log_named_uint("cachedTotalAssets (reported)", reportedTotal);
        emit log_named_uint("actual vault balance", actualTotal);
        emit log_named_uint("phantom (unfunded) amount", phantomAmount);
        emit log_named_uint("Alice shares (worthless)", aliceShares);
        emit log_named_uint("Bob shares (worthless)", bobShares);
    }
}
```

Run with:

```
forge test --match-test test_C01_phantomCachedTotalAssets_vaultInsolvency -vvv
```

The test proves:
1. A strategy with 1,000,000e18 allocated is halted then removed
2. `cachedTotalAssets` stays at 1,000,000e18 while the vault holds 0 real tokens
3. Both Alice and Bob's withdrawals revert because there are no assets to transfer
4. The phantom balance equals the entire removed allocation

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:M/Y:M (1.5)

Recommendation

Consider reducing `cachedTotalAssets` by the stranded allocation before deleting the strategy data.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** made a business decision to acknowledge this finding and not alter the contracts.

Client Note:
We acknowledge this finding but accept the risk, as the described behavior is tied to an intentional emergency admin path rather than a user-triggerable vulnerability. It requires an explicit admin action to halt and emergency-remove a strategy through a high-threshold admin multisig with robust policies. This flow is by design and is intended for incident response: removing broken strategies and migrating positions to healthy ones through deployment of new contracts, upgrades, admin actions, and position rebalancing. In this context, not affecting total assets is a desired outcome, since the exchange rate is treated as an invariant and should not change unless a real loss has occurred. If such an action were ever needed, we would manually migrate funds to a new strategy, attach that strategy to the vault, and initiate an unbacked allocation so the new strategy can take over and report the same total assets, with no loss or profit reported.

7.6 MINIMUM TRANSACTION AMOUNTS NOT REFLECTED IN MAXDEPOSIT AND MAXWITHDRAW VIEW FUNCTIONS

// INFORMATIONAL

Description

The vault enforces `minTxDepositAmount` in `_deposit()` and `minTxWithdrawAmount` in `withdraw()/redeem()`, but the ERC-4626 view functions `maxDeposit()` and `maxWithdraw()` do not account for these minimums. A user can call `maxDeposit()`, receive a value below `minTxDepositAmount`, attempt the deposit, and have it revert.

Code Location

`StandardVaultHelperLib.sol`, `calcMaxDeposit()` ignores minimum:

```
function calcMaxDeposit(uint256 maxGlobalDepositLimit, uint256 totalAssets_) internal pure returns (u
    if (maxGlobalDepositLimit == 0) {
        return 0;
    }
    if (totalAssets_ >= maxGlobalDepositLimit) {
        return 0;
    }
    return maxGlobalDepositLimit - totalAssets_;
    // Can return a value below minTxDepositAmount
    // Depositing this amount would revert in _deposit()
}
```

`StandardVaultHelperLib.sol`, `calcMaxWithdraw()` ignores minimum:

```
function calcMaxWithdraw(
    uint256 ownerShares,
    uint256 maxTxWithdrawLimit,
    uint256 totalAssets_,
    uint256 totalSupply_
) internal pure returns (uint256) {
    if (maxTxWithdrawLimit == 0) {
        return 0;
    }
    uint256 ownerAssets =
        ConversionLib.calcConvertToAssets(ownerShares, totalSupply_, totalAssets_, Math.Rounding.Floo
    return ownerAssets > maxTxWithdrawLimit ? maxTxWithdrawLimit : ownerAssets;
    // Can return a value below minTxWithdrawAmount
    // Withdrawing this amount would revert in withdraw()
}
```

`ConcreteStandardVaultImpl.sol`, `_deposit()` enforces the minimum that the view ignores:

```
uint256 minDepositAmount = SVLib.fetch().minTxDepositAmount;
require(assets >= minDepositAmount, MinimumDepositAmountNotMet(msg.sender, assets, minDepositAmount))
```

`ConcreteStandardVaultImpl.sol`, `withdraw()` enforces the minimum that the view ignores:

```
uint256 minWithdrawAmount = SVLib.fetch().minTxWithdrawAmount;
require(assets >= minWithdrawAmount, MinimumWithdrawAmountNotMet(msg.sen
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:L/D:N/Y:N (1.3)

Recommendation

Consider returning zero from `maxDeposit()` and `maxWithdraw()` when the computed value falls below the respective minimum, so that integrators correctly interpret the vault as having no valid deposit or withdrawal available at that size.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this finding in the specified commit by passing the configured minimum transaction amounts into all four `calcMax*` helper functions and returning zero when the computed maximum falls below the respective minimum threshold, ensuring that `maxDeposit`, `maxMint`, `maxWithdraw`, and `maxRedeem` accurately reflect amounts that `deposit()` and `withdraw()` would accept.

Client Note:
Updated the max preview functions to reflect the min amounts (return zero if min amount is lower than computed max).

Remediation Hash

9e725ef892d5ed6d299d3bd7d4950a04ca3764dc

7.7 DEFAULT_ADMIN_ROLE RENOUNCEMENT PERMANENTLY BRICKS FEEACCOUNTANT

// INFORMATIONAL

Description

The `FeeAccountant.initialize()` function grants both `DEFAULT_ADMIN_ROLE` and `DISTRIBUTOR_ROLE` to a single admin address. Because `DEFAULT_ADMIN_ROLE` is its own admin by OpenZeppelin's default configuration, if this admin calls `renounceRole(DEFAULT_ADMIN_ROLE, admin)`, the role can never be re-granted.

Code Location

`FeeAccountant.sol`, `initialize()` grants admin to a single address:

Copy Code

```
function initialize(address admin_, address vault_) external initializer {
    if (admin_ == address(0)) revert ZeroAdmin();
    if (vault_ == address(0)) revert ZeroVault();
    __AccessControlEnumerable_init();
    __UUPSUpgradeable_init();
    vault = IERC4626(vault_);
    _grantRole(DEFAULT_ADMIN_ROLE, admin_);
    // Single holder of DEFAULT_ADMIN_ROLE
    // DEFAULT_ADMIN_ROLE is its own admin (OZ default)
    _grantRole(DISTRIBUTOR_ROLE, admin_);
}
```

`FeeAccountant.sol`, functions gated by `DEFAULT_ADMIN_ROLE` that become permanently inaccessible:

Copy Code

```
function addRecipient(address recipient) external onlyRole(DEFAULT_ADMIN_ROLE) {
    // Permanently blocked after admin renouncement
    // ...
}

function removeRecipient(address recipient) external onlyRole(DEFAULT_ADMIN_ROLE) {
    // Permanently blocked
    // ...
}

function rescueToken(address token, address to, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE)
    // Permanently blocked
    // ...
}

function _authorizeUpgrade(address) internal override onlyRole(DEFAULT_ADMIN_ROLE) {}
// Permanently blocked -- UUPS upgrades impossible
```

OpenZeppelin's `AccessControl` (inherited), `renounceRole()` available to any holder:

Copy Code

```
// From OZ AccessControl -- not overridden in FeeAccountant:
function renounceRole(bytes32 role, address callerConfirmation) public virtual {
    // ...
    _revokeRole(role, callerConfirmation);
    // No check for last holder -- role can be permanently emptied
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.5)

Recommendation

Consider overriding `renounceRole()` in `FeeAccountant` to prevent the last `DEFAULT_ADMIN_ROLE` holder from renouncing.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** made a business decision to acknowledge this finding and not alter the contracts.

Client Note:
We acknowledge this risk. However, we consider the probability of renouncing the final Admin role to be almost zero, given that it is held by a multisig wallet with robust policies. In addition, the DISTRIBUTOR_ROLE, which is assigned to a separate address for normal operations, would also have to be renounced. Even in the astronomically unlikely event that all roles were somehow accidentally renounced, the only funds at risk would be uncollected fees.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.