// Security Assessment

04.28.2026 - 05.04.2026

Earn 1.4 & AssetCx 1.2

Blueprint Finance

HALBORN

Earn 1.4 & AssetCx 1.2 - Blueprint Finance

Prepared by: HALBORN

Last Updated 05/19/2026

Date of Engagement: April 28th, 2026 - May 4th, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	0	0	1	9

TABLE OF CONTENTS

1. Summary

`Blueprint Finance` engaged Halborn to conduct a security assessment on their smart contracts beginning on April 28th, 2026 and ending on May 4th, 2026. The security assessment was scoped to the smart contracts provided in the `Blueprint-Finance/earn-v2-core` and `Blueprint-Finance/ASSETcx` Github repositories, provided to the Halborn team. Commit hash and further details can be found in the Scope section of this report.

The review covers the diff introduced across two repositories. The Earn V2 core changes introduce deposit lock mechanics with an optional time-decaying early unlock fee, partial epoch settlement for the async vault, and a vault configuration refactor, alongside modifications to existing hooks and interfaces to align with the updated `IHook` signatures. The `AssetCX` repository changes are minimal, migrating the whitelist hook and vault implementation to the new `IHook` interface and introducing a dedicated contract identifier for the AssetCX vault type.

2. ASSESSMENT SUMMARY

`Halborn` was provided with 5 days for this engagement and assigned a full-time security engineer to assess the security of the smart contracts in scope. The assigned engineer possess deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objective of this assessment is to:

- Identify potential security issues within the diff introduced in `Blueprint-Finance Earn Core V2` and `Blueprint AssetCx` smart contracts.
- Ensure that smart contract of `Blueprint-Finance Earn Core V2` and `Blueprint AssetCx` functions operate as intended.

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

- `Restrict lock creation to same-address deposits or require explicit receiver consent, and enforce a meaningful minimum deposit amount per lock entry.`
- `Track cumulative deposited amounts rather than current holdings to prevent cap bypass through withdrawals and transfers.`
- `Enforce a non-zero minimum and a protocol-defined maximum bound on the lock count configuration to prevent unbounded array growth and block gas limit violations.`

3. CAVEATS

This assessment only covers the changes introduced between commit `a90622a` and commit `3e950d7` in `Blueprint-Finance/earn-v2-core` Github repository, and the changes introduced between commit `d636dae` and commit `06cc30f` in `Blueprint-Finance/ASSETcx` Github repository. All other parts of the codebase are treated as black boxes, and it is assumed that any required security mechanisms are properly implemented elsewhere in the system. This includes, but is not limited to, input validation, business logic enforcement, and access control.

Differential audits focus on identifying security risks introduced or modified in a specific set of changes. While some components added in the diff may interact with each other or with existing logic, these interactions are only reviewed when they are explicitly visible within the diff itself. Implicit assumptions or cross-component dependencies may fall outside the scope unless clearly surfaced through changes or supporting context.

For a more complete understanding of the protocol's security posture and the behavior of interconnected components, a full-scope assessment is recommended.

4. TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

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

- Research into the architecture and purpose of the `Blueprint-Finance Earn Core V2` and `Blueprint AssetCx` contracts.
- Manual code review and walkthrough of the `Blueprint-Finance Earn Core V2` and `Blueprint AssetCx` in-scope contracts.
- Manual assessment of critical `Solidity` variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local deployment and testing with (`Foundry`, `Remix IDE`).

5. 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.

5.1 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:

$$E = \prod m_e$$

5.2 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:

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

5.3 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

6. SCOPE

REPOSITORIES

(a) Repository: earn-v2-core

(b) Assessed Commit ID: 3e950d7

(c) Items in scope:

- src/common/ERC20UpgradeableWithTransferHook.sol
- src/common/UpgradeableVault.sol
- src/implementation/ConcreteAsyncVaultImpl.sol
- src/implementation/ConcreteStandardVaultImpl.sol
- src/interface/IConcreteAsyncVaultImpl.sol
- src/interface/IConcreteStandardVaultImpl.sol
- src/interface/IHook.sol
- src/lib/AsyncVaultHelperLib.sol
- src/lib/Hooks.sol
- src/lib/StateSetterLib.sol
- src/lib/WithdrawLib.sol
- src/oracle/OffchainHurdleRateOracle.sol
- src/periphery/hooks/DepositLockHook.sol
- src/periphery/hooks/DepositLockWithFeeHook.sol
- src/periphery/hooks/HookContainer.sol
- src/periphery/hooks/UserDepositCapHook.sol
- src/periphery/hooks/WhitelistUserDepositHook.sol
- src/periphery/interface/IUserDepositCapHook.sol

Out-of-Scope: Third party dependencies and economic attacks. Furthermore, this assessment was limited to the differential changes between commit a90622a and commit 3e950d7. Components outside of the reviewed diff, pre-existing logic not modified by the changes, system-wide interactions not directly surfaced in the diff context, and external assumptions regarding input validation, business rules, and access controls were explicitly out of scope.

(a) Repository: ASSETcx

(b) Assessed Commit ID: 06cc30f

(c) Items in scope:

- src/hooks/WhitelistHook.sol
- src/implementation/ConcreteAsyncVaultImpl.sol

Out-of-Scope: Third party dependencies and economic attacks. Furthermore, this assessment was limited to the differential changes between commit d636dae and commit 06cc30f. Components outside of the reviewed diff, pre-existing logic not modified by the changes, system-wide

interactions not directly surfaced in the diff context, and external assumptions regarding input validation, business rules, and access controls were explicitly out of scope.

REMEDIATION COMMIT ID:

- dead10e
- beacca0
- 8db5d84

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

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	1

INFORMATIONAL
9

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PER-USER CAP ENFORCEMENT BYPASSED VIA SECONDARY TOKEN TRANSFERS	LOW	RISK ACCEPTED - 05/18/2026
UNCAPPED ARRAY GROWTH VIA ADMINISTRATIVE PARAMETER MISCONFIGURATION CAUSES GAS DOS	INFORMATIONAL	SOLVED - 05/17/2026
ANY DEPOSITOR CAN FILL VICTIM'S LOCK SLOTS VIA ARBITRARY RECEIVER DEPOSIT	INFORMATIONAL	ACKNOWLEDGED - 05/18/2026
TRANSIENT REENTRANCY GUARD SILENTLY FAILS ON CHAINS WITHOUT EIP-1153 SUPPORT	INFORMATIONAL	ACKNOWLEDGED - 05/18/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PARTIAL SETTLEMENT FUNCTION APPLIES NON-UNIFORM PRICING WITHIN A SINGLE EPOCH LIFECYCLE	INFORMATIONAL	ACKNOWLEDGED - 05/18/2026
MINIMUM LOCK PERIOD MISCONFIGURATION LOCKS OWNER OUT OF RE-ENABLING DEPOSIT LOCKS	INFORMATIONAL	ACKNOWLEDGED - 05/18/2026
MISSING REQUIRED EVENT EMISSION ON SUCCESSFUL ASYNC WITHDRAWAL ACCEPTANCE	INFORMATIONAL	ACKNOWLEDGED - 05/18/2026
CAPACITY VIEW FUNCTIONS RETURN NON-ZERO VALUES WHEN PROTOCOL OPERATIONS ARE SUSPENDED	INFORMATIONAL	SOLVED - 05/14/2026
DUPLICATE ENTRIES IN ORDERED STRATEGY LIST CORRUPT WITHDRAWAL ACCOUNTING	INFORMATIONAL	ACKNOWLEDGED - 05/18/2026
VAULT IMPLEMENTATION UPGRADE PATH MISSING OVERRIDE LEAVES PROTOCOL-SPECIFIC STATE UNINITIALIZED	INFORMATIONAL	SOLVED - 05/14/2026

8. FINDINGS & TECH DETAILS

8.1 PER-USER CAP ENFORCEMENT BYPASSED VIA SECONDARY TOKEN TRANSFERS

// LOW

Description

The `UserDepositCapHook._checkDepositLimit()` enforces per-user deposit caps by checking `VAULT.balanceOf(receiver)` at the time of deposit, converting shares to assets via `VAULT.convertToAssets()`. This approach tracks current holdings rather than cumulative deposits, which means that any reduction in the receiver's share balance, whether by direct transfer, withdrawal, or redemption, immediately restores their apparent capacity under the cap and allows them to deposit again up to the full cap amount. The same single user can repeatedly deposit → transfer out → deposit again, bypassing the intended limit entirely.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

1. Track cumulative deposits instead of (or in addition to) current holdings. Add a `mapping(address => uint256) public cumulativeDeposited` that is incremented on `postDeposit/postMint` and is not decremented on withdrawal. The cap should then be checked against `cumulativeDeposited[receiver] + additionalAssets`.
2. Implement a transfer hook that either prevents transfers from a capped user or adjusts the cumulative tracking on transfer. Without this, splitting deposits across sybil wallets still bypasses the cap.

Remediation Comment

RISK ACCEPTED: The `Blueprint Finance` team accepted the risk of this finding, stating:

> UserDepositCapHook enforces the cap on the receiver at preDeposit / preMint time and intentionally does not implement preTransfer / postTransfer. The hook was built for the basic-user / first-time-buyer scenarios where a retail-style per-wallet entry cap is useful (e.g. early access deposits, fair-launch caps); it was never intended to be a Sybil-resistant per-identity ceiling.
> We are aware of the limitation and accept it as a design trade-off.

8.2 UNCAPPED ARRAY GROWTH VIA ADMINISTRATIVE PARAMETER MISCONFIGURATION CAUSES GAS DOS

// INFORMATIONAL

Description

The `setMaxLocksPerUser()` function in `DepositLockHook` accepts 0 as a valid value, which explicitly disables the per-user lock count cap. When `maxLocksPerUser = 0`, there is no upper bound on how many lock entries a user can accumulate in their `_locks[user]` storage array. Each deposit creates a new lock entry, and the sorted insertion (`_appendLock()`), expiry cleanup (`_cleanupExpired()`), and prefix removal (`_removePrefix()`) operations are all O(N) in storage reads/writes over this array. When the array grows large enough, every deposit, withdrawal, redeem, and transfer involving that user will consume an unbounded and ever-growing amount of gas, eventually making these operations prohibitively expensive or impossible (gas limit exceeded).

Copy Code

```
function _appendLock(address user, uint256 shares) internal {
    //...
    uint256 maxLocks = maxLocksPerUser;
    if (maxLocks != 0 && pending >= maxLocks) {
        revert MaxLocksExceeded(user, pending, maxLocks);
    }

    //...
    if (pending == 0 || unlockTs >= locks[pending - 1].unlockTimestamp) {
        locks.push(newLock);
    } else {
        locks.push(newLock);
        //...
    }
    //...
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

Enforce a non-zero minimum and a protocol-defined maximum on `maxLocksPerUser` to prevent unbounded lock array growth that would cause O(N) operations to exceed block gas limits, and use a sentinel value such as `type(uint256).max` only if unlimited locks are explicitly required with documented gas consequences.

Remediation Comment

SOLVED: The `Blueprint Finance` team solved the issue in the specified commit id by enforcing both a non-zero minimum and a protocol-defined upper ceiling on the lock count configuration, with the setter now reverting on both out-of-range values, stating:

> Even though reaching a problematic array size is practically infeasible when the vault's `minDeposit` is configured (the attacker pays `N × minDeposit` to the victim as redeemable shares plus `N × gas` of pure cost), we added a defensive ceiling in a separate commit on top of the previous one — `MAX_LOCKS_PER_USER_CEILING = 500`, with `setMaxLocksPerUser` now also reverting with `MaxLocksPerUserAboveCeiling(requested, ceiling)` above that value. The number was picked empirically: the worst-case per-user O(N) path at the ceiling `deposit` with adversarial sorted-insert) measures under 1M gas, leaving a >30x margin against a 30M block gas limit on every supported chain. NatSpec, spec, changelog and two boundary tests were updated accordingly.

Remediation Hash

dead10e8b9f84e528005aa129837eedf81ed0060

8.3 ANY DEPOSITOR CAN FILL VICTIM'S LOCK SLOTS VIA ARBITRARY RECEIVER DEPOSIT

// INFORMATIONAL

Description

The `DepositLockHook._appendLock()` function creates a lock entry for the receiver on every `postDeposit()` / `postMint()` hook invocation. Because the ERC-4626 `deposit(uint256 assets, address receiver)` function allows any caller to specify an arbitrary receiver, any account can deposit the vault's minimum required amount and direct the resulting lock to any victim address. When a victim's `_locks` array reaches `maxLocksPerUser` (default 10), every subsequent deposit attempt for that address reverts with `MaxLocksExceeded`, permanently blocking new deposits until enough existing locks expire naturally.

Copy Code

```
308 function _appendLock(address user, uint256 shares) internal {
309     //...
310     uint256 maxLocks = maxLocksPerUser;
311     if (maxLocks != 0 && pending >= maxLocks) {
312         revert MaxLocksExceeded(user, pending, maxLocks);
313     }
314
315     //...
316     if (pending == 0 || unlockTs >= locks[pending - 1].unlockTimestamp) {
317         locks.push(newLock);
318     } else {
319         locks.push(newLock);
320         //...
321     }
322     //...
323 }
```

BVSS

AO:A/AC:M/AX:M/R:F/S:U/C:N/A:M/I:N/D:N/Y:N (0.6)

Recommendation

Restrict lock creation to deposits where the sender and receiver are the same address or require explicit receiver consent to prevent imposing locks on third parties, and enforce a meaningful minimum deposit amount per lock entry to prevent dust deposits from inflating the lock queue toward the maximum cap.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance` team acknowledged the risk of this finding, stating:

> We disputed the original Medium severity and the auditor downgraded the finding to Informational based on our response. Summarising the rationale for the record:
> - Attacker pays, victim receives value. deposit(assets, receiver) debits the caller and mints shares to the receiver. To fill all maxLocksPerUser slots the attacker has to transfer minDeposit ×

maxLocksPerUser worth of assets to the victim, who ends up holding them as fully redeemable shares. Not a dust attack — the attacker is subsidizing the victim.
- Bounded, recoverable impact. Only new deposits to the victim's address are blocked, and only until the next lock expires (≤ depositLockDuration). Existing shares remain transferable, withdrawable, and redeemable — _requireUnlocked only checks shares > unlocked. No funds are lost or frozen.
Mitigation belongs in the documented config model. DepositLockHook explicitly defers dust / minimum-deposit enforcement to the vault: "Minimum deposit / dust limits are expected to be enforced by the vault (or another hook), not here." Operators control minDeposit on the vault and maxLocksPerUser / depositLockDuration on the hook — these are the intended risk levers.
- BVSS scoring nit. AX:L overstates attack cost: the cost is bounded below by the operator-set minDeposit, not structurally low.

8.4 TRANSIENT REENTRANCY GUARD SILENTLY FAILS ON CHAINS WITHOUT EIP-1153 SUPPORT

// INFORMATIONAL

Description

`UpgradeableVault` was migrated from `ReentrancyGuardUpgradeable` to `ReentrancyGuardTransient`, which relies on the `TSTORE` and `TLOAD` opcodes introduced in EIP-1153 and activated on Ethereum mainnet via the Cancun upgrade in March 2024 and on Arbitrum via ArbOS 32 in November 2024. On any EVM-compatible chain that has not adopted Cancun, these opcodes resolve to an INVALID instruction, causing all nonReentrant-guarded vault functions to revert unconditionally, rendering the vault non-operational on unsupported chains. Current deployment targets Ethereum L1 and Arbitrum, both support EIP-1153. Any future deployment to a chain outside this set requires explicit EIP-1153 verification as part of the pre-deployment process.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:L/I:N/D:N/Y:N (0.6)

Recommendation

Verify EIP-1153 support on every new target deployment chain before deploying, and document the supported chain list as part of the protocol's deployment prerequisites.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance` team acknowledged the risk of this finding, stating:

> All target deployment chains for this protocol (Ethereum, Polygon PoS, BSC, Arbitrum, Optimism / Base / Mode, Linea, Scroll, zkSync Era, Avalanche C-Chain) activated EIP-1153 between March and June 2024. The build is pinned to evm_version = "cancun" in foundry.toml, so the bytecode could not run on a pre-Cancun chain even if one were targeted — the issue would surface immediately on the first call, not as a subtle reentrancy weakness. No change required.

8.5 PARTIAL SETTLEMENT FUNCTION APPLIES NON-UNIFORM PRICING WITHIN A SINGLE EPOCH LIFECYCLE

// INFORMATIONAL

Description

`AsyncVaultHelperLib.processPartialEpochRequest()` settles a user's closed-epoch request at the live spot price at call time rather than a price locked for the epoch. The epoch is not marked as processed after partial settlement, allowing remaining users to be settled later via `processEpoch()` at a different price. A user whose request is partially settled has no ability to cancel as `cancelRequest()` only accepts open epochs, meaning they cannot object to the price applied. The `WITHDRAWAL_MANAGER` can time calls to selectively favor or disadvantage individual users within the same epoch.

BVSS

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

Recommendation

Lock and record the settlement price on the first call to `processPartialEpochRequest()` for a given epoch and enforce that all subsequent partial settlements and the final `processEpoch()` call use the same locked price.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance` team acknowledged the risk of this finding, stating:

> Design decision, no change required.
> Using the live per-call exchange rate (rather than a per-epoch snapshot) is intentional and documented in doc/spec/PartialEpochProcessing.md. The behaviour is part of the async withdrawal design and we keep it as-is.

8.6 MINIMUM LOCK PERIOD MISCONFIGURATION LOCKS OWNER OUT OF RE-ENABLING DEPOSIT LOCKS

// INFORMATIONAL

Description

`DepositLockWithFeeHook.setMinLockPeriod()` skips its cross-parameter validation when `depositLockDuration == 0`. This allows the owner to set `minLockPeriod` to any arbitrary value. When the owner later attempts to re-enable locking via `setDepositLockDuration(newDuration)`, the call reverts if `minLockPeriod >= newDuration`, blocking lock re-enablement entirely until the owner explicitly resets `minLockPeriod` to zero first.

Copy Code

```
142 function setMinLockPeriod(uint64 newPeriod) external onlyOwner {
143     uint64 duration = depositLockDuration;
144     if (newPeriod > 0 && duration > 0 && newPeriod >= duration) {
145         revert MinLockPeriodExceedsDuration(newPeriod, duration);
146     }
147     uint64 previous = minLockPeriod;
148     minLockPeriod = newPeriod;
149     emit MinLockPeriodSet(previous, newPeriod);
150 }
```

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:L/I:N/D:N/Y:N (0.1)

Recommendation

Enforce that `minLockPeriod` cannot be set to a value that would conflict with any valid future lock duration, regardless of the current `depositLockDuration` value.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance` team acknowledged the risk of this finding, stating:

> Documentation update only.
> We accept the factual observation: when depositLockDuration == 0, setMinLockPeriod skips its cross-parameter check (because the invariant minLockPeriod < depositLockDuration is vacuously satisfied while no locks can be created), so the owner can leave the contract in a state where a subsequent setDepositLockDuration(newDuration <= minLockPeriod) will revert with MinLockPeriodExceedsDuration.
> We classify this as an owner-only operational footgun rather than a code defect:
> - Both setters are onlyOwner; no adversarial vector. Recovery is a two-step admin action (setMinLockPeriod(0) then setDepositLockDuration(newDuration)), no funds are at risk and no state is unrecoverable.
> - The revert (MinLockPeriodExceedsDuration(minPeriod_, newDuration)) carries both conflicting values, so the cause is self-evident from the failing transaction.

- The proposed mitigation — "enforce that minLockPeriod cannot conflict with any valid future lock duration regardless of the current depositLockDuration" — is not implementable without choosing an arbitrary maximum for depositLockDuration, which is currently uint64 and intentionally unbounded.
Changes made:
- NatSpec on DepositLockWithFeeHook.setMinLockPeriod and DepositLockWithFeeHook.setDepositLockDuration updated to document the re-enable-after-disable ordering constraint explicitly, so operators have an on-source reference for the recovery procedure. No behaviour change; documentation only.

8.7 MISSING REQUIRED EVENT EMISSION ON SUCCESSFUL ASYNC WITHDRAWAL ACCEPTANCE

// INFORMATIONAL

Description

The ERC-4626 specification mandates that a `Withdraw` event must be emitted on every successful invocation of `withdraw()` and `redeem()`. When `ConcreteAsyncVaultImpl`'s withdrawal queue is active (`isQueueActive == true`), the overridden `_executeWithdraw()` queues the request and bypasses `withdraw()`, which is the function that both transfers assets and emits the `Withdraw` event. As a result, successful calls to `withdraw()` or `redeem()` in async mode complete without emitting the required `Withdraw` event, violating a must requirement of the ERC-4626 standard.

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Emit the ERC-4626 `Withdraw` event in `ConcreteAsyncVaultImpl._executeWithdraw()` when the queue is active, immediately after the `QueuedWithdrawal` emit, to satisfy the MUST requirement of the spec despite assets not yet being transferred.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance` team acknowledged the risk of this finding, stating:

> Design decision, no change required.
> The deviation from the ERC-4626 Withdraw MUST-emit clause is inherent to the async withdrawal pattern and was not introduced by this audit's changes — ConcreteAsyncVaultImpl has shipped with this behavior since the first release. The vault implements the request → settle → claim lifecycle (closer to ERC-7540 semantics than to synchronous ERC-4626), and each phase already has a dedicated event: QueuedWithdrawal on _executeWithdraw, EpochProcessed / PartialEpochRequestProcessed on settlement, and RequestClaimed on claim.
> We do not plan to emit Withdraw inside _executeWithdraw when the queue is active, as the auditor recommends, because the event's semantics ("assets transferred out of the vault, shares burned") are not satisfied at queue time: assets remain in the vault, shares are escrowed (not burned) until processEpoch / processPartialEpochRequest, and the request can still be cancelled via cancelRequest. Emitting Withdraw at that point would create double-counting in indexers and dashboards that sum withdrawal volume from Withdraw events, and would orphan the event if the request is later cancelled. The existing event set captures the full async lifecycle without these issues.

8.8 CAPACITY VIEW FUNCTIONS RETURN NON-ZERO VALUES WHEN PROTOCOL OPERATIONS ARE SUSPENDED

// INFORMATIONAL

Description

The ERC-4626 specification explicitly requires that `maxDeposit()`, `maxMint()`, `maxWithdraw()`, and `maxRedeem()` return 0 when the corresponding vault operation is currently disabled. The `ConcreteStandardVaultImpl` vault uses the `whenNotPaused` modifier on deposit, mint, withdraw, and redeem, meaning all four operations revert when the vault is paused. However, the corresponding `maxDeposit()`, `maxMint()`, `maxWithdraw()`, and `maxRedeem()` view functions do not check the paused state and will return non-zero values even when all operations are disabled. This violates the ERC-4626 invariant that max functions must accurately reflect the current operational limits of the vault.

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add a pause state check at the beginning of each max function to return 0 when the vault is paused.

Remediation Comment

SOLVED: The `Blueprint Finance` team solved the issue in the specified commit id by following the mentioned recommendation, stating:

> We agree with the finding. The ERC-4626 spec explicitly requires maxDeposit / maxMint / maxWithdraw / maxRedeem to return 0 when the corresponding operation is disabled (even temporarily), and whenNotPaused on deposit / mint / withdraw / redeem falls under that clause.
> Changes made:
> ConcreteStandardVaultImpl.sol: added if (paused()) return 0; as the first line of maxDeposit, maxMint, maxWithdraw, and maxRedeem. ConcreteAsyncVaultImpl inherits these overrides and therefore picks up the same behavior automatically.
> The check is intentionally limited to paused(). For ConcreteAsyncVaultImpl in queue mode (isQueueActive == true), withdraw / redeem are not disabled — they enqueue requests rather than settling synchronously — so maxWithdraw / maxRedeem continue to report non-zero values in that mode, which is the correct semantics.

Remediation Hash

beacca0f7261fd6454e0e2ddda03a4f8ac8f400b

8.9 DUPLICATE ENTRIES IN ORDERED STRATEGY LIST CORRUPT WITHDRAWAL ACCOUNTING

// INFORMATIONAL

Description

The `setDeallocationOrder()` function in `StateSetterLib` validates that each strategy in the provided order array is registered and active, but does not check for duplicate entries. A deallocation order array containing the same strategy address multiple times (e.g., [strategyA, strategyA]) passes validation and is stored directly. When `WithdrawLib.executeWithdrawFromStrategies()` subsequently iterates this order, the same strategy is visited twice. After the first iteration withdraws from strategyA and decrements `$.strategyData[strategyA].allocated`, a second iteration attempts to withdraw again from the same strategy. If the first iteration reduced allocated to 0, the second iteration will underflow (`allocated -= withdrawn`) causing the entire withdrawal to revert, effectively blocking all withdrawals until the deallocation order is corrected.

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add duplicate detection in `setDeallocationOrder()` to reject arrays with repeated strategy addresses.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance` team acknowledged the risk of this finding, stating:

> Design decision, no change required.
> We accept the factual observation that setDeallocationOrder does not reject duplicate entries, but we disagree with the claim that this corrupts withdrawal accounting and we keep the current behavior intentionally.
> Why duplicates do not cause the described underflow:
> WithdrawLib.executeWithdrawFromStrategies reads strategy.maxWithdraw() at the start of every iteration and bounds the withdrawal request to that value. After the first visit to a duplicated strategy drains its position, the strategy's maxWithdraw() reflects the post-drain state on the second visit (the loop is synchronous within a single transaction, so the strategy's view is already updated). withdrawAmount then evaluates to min(0, desiredAssets) = 0, the if (withdrawAmount > 0) branch is skipped, onWithdraw is never called, and allocated is not decremented. The second iteration is a silent no-op, not an underflow.
> For allocated -= actualWithdrawn to underflow, actualWithdrawn would need to exceed allocated, which requires the strategy to report a balance that exceeds the vault's allocated tracking (e.g., unaccrued yield, direct token donations to the strategy, or a strategy that violates the actualWithdrawn ≤ maxWithdraw() invariant enforced by BaseStrategy.onWithdraw). None of those causes depend on duplicates; they are independent accounting-synchronization concerns, and the

withYieldAccrual modifier on all withdrawal paths keeps allocated in sync with strategy state under normal operation.
Allowing duplicates is a deliberate design choice. It was already acknowledged in previous audits.

8.10 VAULT IMPLEMENTATION UPGRADE PATH MISSING OVERRIDE LEAVES PROTOCOL-SPECIFIC STATE UNINITIALIZED

// INFORMATIONAL

Description

`ConcreteAsyncVaultImpl` does not override the `_upgrade()` virtual function from its parent `ConcreteStandardVaultImpl`. When a vault proxy is upgraded from a standard vault implementation (ID=110200) to the async vault implementation (ID=210200), the factory calls `VaultProxy.upgradeToAndCall(newImpl, abi.encodeCall(IUpgradeableVault.upgrade, (newVaultImpl, data)))`. This invokes `upgrade()` in `UpgradeableVault`, which calls `_upgrade()`. Because `ConcreteAsyncVaultImpl` has no `_upgrade()` override, the parent's `ConcreteStandardVaultImpl.upgrade()` runs instead, which only performs `RoleMigrationLib.migrateLegacyAdminRoles(msg.sender)`. The critical `StateInitLib.stateInitAsyncVaultImpl()` function, which sets `$.isQueueActive = true`, `$.latestEpochID = 1`, `$.unwindCostCapBP = 500`, and configures `WITHDRAWAL_MANAGER` and `PRIORITY_WITHDRAWAL_EXECUTOR` role admins, is **never called** during the upgrade path.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Override `_upgrade()` in `ConcreteAsyncVaultImpl` to initialize async-specific state, Alternatively, expose a dedicated post-upgrade initialization function with appropriate access controls that the factory calls via the `data` parameter of `upgradeToAndCall()`.

Remediation Comment

SOLVED: The `Blueprint Finance` team solved the issue in the specified commit id by following the mentioned recommendation, stating:

> We agree with the finding. ConcreteAsyncVaultImpl previously inherited _upgrade from ConcreteStandardVaultImpl, which only performs legacy role migration. As a result, a Standard → Async upgrade routed through ConcreteFactory.upgrade would leave isQueueActive, latestEpochID, unwindCostCapBP, and the WITHDRAWAL_MANAGER / PRIORITY_WITHDRAWAL_EXECUTOR role admins uninitialised — rendering the upgraded async vault inoperable until those slots were set out-of-band.
> Changes made:
> - StateInitLib.stateUpgradeAsyncVaultImpl() (src/lib/StateInitLib.sol, new function): single library entry point for the upgrade path. Calls RoleMigrationLib.migrateLegacyAdminRoles(msg.sender) (preserving the legacy role-migration step the parent used to perform), then runs the async-specific state init iff latestEpochID == 0. latestEpochID == 0 is a safe "never initialised" sentinel: the init unconditionally sets it to 1, and the value only ever increases via closeEpoch. Async → Async

upgrades (e.g. minor version bumps) therefore skip the init and preserve operator-tuned config (unwindCostCapBP) and in-flight epoch state. The function is public and runs in the proxy's storage context via DELEGATECALL.
- ConcreteAsyncVaultImpl._upgrade (src/implementation/ConcreteAsyncVaultImpl.sol): added an override carrying only the whenNotPaused modifier and a single delegated call to StateInitLib.stateUpgradeAsyncVaultImpl(). This bytecode-minimising shape was chosen because ConcreteBridgedAsyncVaultImpl (which inherits from ConcreteAsyncVaultImpl) was already against the EIP-170 24,576-byte limit prior to this audit; consolidating the upgrade logic in the library keeps both contracts below the limit (post-change runtime margins: ConcreteAsyncVaultImpl +173 B, ConcreteBridgedAsyncVaultImpl +65 B).
ConcreteBridgedAsyncVaultImpl inherits from ConcreteAsyncVaultImpl and therefore picks up the override automatically. The admin still needs to explicitly grant the new async roles (WITHDRAWAL_MANAGER, PRIORITY_WITHDRAWAL_EXECUTOR) to operators post-upgrade; the override only configures the role admin relationships, not the grants — this is intentional because the target operator addresses are deployment-specific.

Remediation Hash

8db5d844912e8b3b545e7028178bef19e2678d1d
