CANTINA

Concrete Finance: Earn v2 Core

Security Review

Cantina Managed review by:
Phaze, Lead Security Researcher
Kankodu, Security Researcher

March 3, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Nov 4th to Nov 9th the Cantina team conducted a review of earn-v2-core on commit hash 6edba8d7. The team identified a total of **14** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	6	3	3
Gas Optimizations	2	2	0
Informational	6	3	3
Total	**14**	**8**	**6**

2.1 Scope

The security review had the following components in scope for earn-v2-core on commit hash 6edba8d7:

```
src
├── common
│   └── UpgradeableVault.sol
├── factory
│   ├── ConcreteFactory.sol
│   └── VaultProxy.sol
├── implementation
│   ├── ConcreteAsyncVaultImpl.sol
│   ├── ConcreteBridgedVaultImpl.sol
│   ├── ConcretePredepositVaultImpl.sol
│   └── ConcreteStandardVaultImpl.sol
├── interface
│   ├── IAllocateModule.sol
│   ├── IConcreteAsyncVaultImpl.sol
│   ├── IConcreteBridgedAsyncVaultImpl.sol
│   ├── IConcreteFactory.sol
│   ├── IConcretePredepositVaultImpl.sol
│   ├── IConcreteStandardVaultImpl.sol
│   ├── IConcreteTokenizedVault.sol
│   ├── IHook.sol
│   ├── IStrategyTemplate.sol
│   ├── IUpgradeableVault.sol
│   └── IVaultProxy.sol
├── lib
│   ├── AccessControlLib.sol
│   ├── AsyncVaultHelperLib.sol
│   ├── Constants.sol
│   ├── Conversion.sol
│   ├── ERC20Lib.sol
│   ├── Hooks.sol
│   ├── Roles.sol
│   ├── StateInitLib.sol
│   ├── StateSetterLib.sol
│   ├── storage
│   │   ├── ConcreteAsyncVaultImplStorageLib.sol
│   │   ├── ConcreteCachedVaultStateStorageLib.sol
│   │   ├── ConcreteFactoryBaseStorageLib.sol
│   │   ├── ConcretePredepositVaultImplStorageLib.sol
│   │   └── ConcreteStandardVaultImplStorageLib.sol
│   │
│   └── Time.sol
├── module
│   └── AllocateModule.sol
└── periphery
    ├── auxiliary
    │   └── TwoWayFeeSplitter.sol
    ├── hooks
    │   └── UserDepositCapHook.sol
    ├── interface
    │   ├── IBaseStrategy.sol
    │   ├── IFeeSplitter.sol
    │   ├── IPredepostVaultOApp.sol
    │   └── IUserDepositCapHook.sol
    ├── lib
    │   ├── BaseStrategyStorageLib.sol
    │   ├── MultisigStrategyStorageLib.sol
    │   ├── PeripheryRolesLib.sol
    │   ├── PositionAccountingLib.sol
    │   ├── PositionAccountingStorageLib.sol
    │   ├── PredepostVaultOAppStorageLib.sol
    │   └── SimpleStrategyStorageLib.sol
    ├── predeposit
    │   ├── PredepostVaultOApp.sol
    │   └── ShareDistributor.sol
    └── strategies
        ├── BaseStrategy.sol
        ├── MultisigStrategy.sol
        └── SimpleStrategy.sol
```

3 Findings

3.1 Low Risk

3.1.1 Unhandled insufficient liquidity problem during withdrawals

Severity: Low Risk

Context: ConcreteStandardVaultImpl.sol#L699-L727

Description: In a standard vault, when a user initiates a withdrawal, multiple checks are performed to ensure the contract holds sufficient funds to support it. When users deposit funds, these are allocated to various strategies. Upon withdrawal, the vault attempts to retrieve the required assets from these strategies one by one, following the deallocation order. However, the maximum amount that can be withdrawn instantly across all strategies may be less than the assets the user needs. This scenario is not currently handled in the code. Instead, it simply fails with an "Insufficient balance" error from the ERC20 contract.

This issue can occur, for example, if a strategy deposits funds into a lending protocol with utilization near 100%. In such cases, even if the strategy supports instant withdrawals, the lending protocol may not be able to provide the liquidity.

Recommendation: Instead of relying on underflow errors that result in hard to read failures, fail with a helpful custom error. After the loop in `_executeWithdraw` if `totalWithdrawableAmount` is still less than `assets`, fail with a helpful error before `_withdraw` gets called.

```diff
+  if (totalWithdrawableAmount < assets) {
+      revert ERC4626InsufficientWithdrawableAssets();
+ }
```

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.

3.1.2 ERC4626 non compliance

Severity: Low Risk

Context: ConcreteStandardVaultImpl.sol#L481-L498

Description: The ERC4626 standard requires `maxDeposit` to account for global and user-specific limits, returning 0 if deposits are disabled. However, in the standard vault, `maxDeposit` and `maxMint` are not overridden and always return `type(uint256).max`, ignoring deposit limits set by the vault manager. Similarly, `maxWithdraw` and `maxRedeem` fail to consider withdrawal limits, violating the standard.

Recommendation: Override `maxDeposit` and `maxMint` to return values based on the configured deposit limits, factoring in global and per-user constraints. Likewise, override `maxWithdraw` and `maxRedeem` to incorporate withdrawal limits for full ERC4626 compliance.

Blueprint Finance: Fixed in cantina.

Cantina Managed: Fix verified.

3.1.3 Upgrade handler receives the wrong old version

Severity: Low Risk

Context: UpgradeableVault.sol#L75

Description: `UpgradeableVault.upgrade()` wraps `_upgrade(oldVersion, newVersion, data)` inside `reinitializer(newVersion)`. The modifier updates `_initialized` to `newVersion` before the function body executes, so `_getInitializedVersion()` already returns the new version. As a result `_upgrade()` receives identical values for both the `oldVersion` and `newVersion` parameters and cannot know which implementation it is migrating away from. This prevents any version-specific migration logic or sanity checks that depend on the previous version.

Recommendation: Cache the current version before applying the `reinitializer()` guard or update the reinitialization flow so `_upgrade()` receives both versions accurately, for example by performing the version assignment inside the function after `_upgrade()` succeeds or by storing the previous version locally before changing `_initialized`.

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.

3.1.4 Strategy withdrawal return value is trusted without verification

Severity: Low Risk

Context: ConcreteStandardVaultImpl.sol#L715

Description: `ConcreteStandardVaultImpl._deallocateFromStrategies()` updates accounting based solely on the value returned by `IStrategyTemplate.onWithdraw()`. Because strategies are external contracts, a faulty or compromised strategy could return an inflated amount while transferring fewer tokens back to the vault, causing share redemptions to be paid with unbacked accounting entries. The current logic never checks the actual vault asset balance delta to confirm the claimed transfer, so accounting drift would go unnoticed until losses accumulate.

Recommendation: Consider measuring the vault's asset balance before and after each `onWithdraw()` call and cap the credited amount to the observed delta. This ensures share accounting only reflects funds that actually arrived from the strategy, even if a strategy misreports its withdrawal results.

Blueprint Finance: We are only adding/allowing audited strategies to be added. We acknowledge and accept the risk.

Cantina Managed: Acknowledged.

3.1.5 Vault address prediction drifts when metadata hash changes

Severity: Low Risk

Context: ConcreteFactory.sol#L149-L178

Description: `ConcreteFactory._computeBytecode()` and `predictVaultAddress()` calculate deterministic vault addresses by hashing `type(VaultProxy).creationCode` together with the constructor arguments. The Solidity compiler appends a metadata hash to that creation code, so any repository change that alters the metadata, such as editing comments or changing unrelated files, causes the compiled value to differ even when the proxy logic is identical. If the factory implementation is later upgraded or redeployed with a newly compiled artifact, the metadata hash in the bytecode constant will change, so the same version/owner parameters will hash to different CREATE2 addresses than those previously communicated. This breaks deterministic address calculation and may invalidate integrations that rely on stable precomputed vault addresses.

Recommendation: Consider building the proxy artifact without embedding the metadata hash (e.g., `bytecode_hash = "none"`) so the creation code remains stable across recompilations, or store a fixed canonical proxy creation bytecode in the factory contract instead of referencing `type(VaultProxy).creationCode` dynamically.

Blueprint Finance: The predicted address is deterministic and consistent for a given deployed factory instance. This matches the intended design: address prediction is guarannteed only within the lifecycle of a specific factory deployment, and not across recompilations or redeployment of the factory contract. We only rely on `predictVaultAddress` before the deployment, as part of the pre-deployment flow (e.g. integrations, off-chain coordination). Once a vault is deployed, we no longer depend on address prediction for that vault.

Cantina Managed: Acknowledged.

3.1.6 Cross-chain migration controls are manually coordinated

Severity: Low Risk

Context: ConcretePredepositVaultImpl.sol#L72-L151

Description/Recommendations:

1. Claims can begin while deposits remain open.

   Both `claimOnTargetChain()` and `batchClaimOnTargetChain()` check that deposit and withdrawal limits are zero when the call executes, but the guards do not permanently lock those limits once migration starts. A manager can therefore re-enable deposits after some users have already burned their shares for bridge claims, producing new supply that will never be exported to the destination chain. The resulting mix of locked and unlocked shares complicates accounting and creates opportunities for dilution. Consider binding deposit and withdrawal locking directly to the migration flag (for example, flipping an irreversible switch when `selfClaimsEnabled` becomes true) or refusing new deposits whenever the system tracks outstanding `lockedShares`.

2. No automated reconciliation between burned and minted shares.

   The source vault records how many shares each user burned via `lockedShares`, yet the bridged vault and distributor blindly trust the amount of inventory that operators mint via `unbackedMint` and transfer manually. Because those steps happen off-chain and in separate transactions, there is no deterministic link between the burned supply and the destination balances, so an operator could mint too many shares, keep a portion of them, or underfund the distributor. Consider orchestrating the migration through a single cross-chain message (or tightly scripted transaction bundle) that carries the canonical total supply and asset amounts taken directly from the source chain, allowing the destination contracts to mint and seed inventory using data that cannot be falsified locally.

3. Destination vault accepts deposits before it is backed.

   `ConcreteBridgedVaultImpl` inherits the default deposit/mint entry points and starts with `maxDeposit` set to `type(uint256).max`. Immediately after `unbackedMint`, the vault has `totalSupply > 0` but `totalAssets == 0`, so the share pricing math lets any early depositor exchange a dust amount of assets for nearly the entire share supply. This can happen before migration operators notice, effectively bricking the `unbackedMint` flow or diluting legitimate bridged holders. Consider disabling deposits, mints, and arbitrary transfers until backing assets are recorded and an explicit ”ready” flag is set as part of the migration script.

4. Migration sequencing depends on manual operator handling.

   Executing the migration requires a long checklist: configure the OApp, freeze deposits/withdrawals, ensure management/performance fees are zero, call `unbackedMint`, invoke `adjustTotalAssets`, run `accrueYield()`, transfer shares to the distributor, and only then enable claims on the source chain. Each step occurs in separate transactions across chains, so a missed or misordered action can corrupt accounting or expose the bridged vault. Consider introducing an automated migration entry point (potentially initiated via a cross-chain message) that sequences the critical operations atomically, and bake in sanity checks such as gating `adjustTotalAssets` behind zero-fee configurations to prevent accidental fee minting during the handover.

Blueprint Finance: We intentionally keep migration operator-orchestrated to preserve flexibility in how and when we bridge the underlying (e.g., tranche-based bridging, delayed settlement, congestion/risk controls). This means we do not enforce a fully deterministic, single-message, end-to-end migration on-chain. We acknowledge and accept the residual risk that comes with this design choice.

That said, we mitigate the operational aspects with strict controls:

Multisig governance: All privileged migration steps (`VAULT_MANAGER` / `VAULT_OWNER` actions) are executed via multisignature wallets, with documented operational guidelines and independent cross-checks (LayerZero peer config, fees set to 0, inventory seeded, rate parity, etc.). Funds safety posture: User funds are not exposed to an unprepared destination state. Claims are only enabled once the target vault/distributor configuration is completed and verified, consistent with the spec's sequencing. Atomic setup + automatic reverts: Critical L2 setup is executed in atomic transactions with built-in verification (e.g., exchange-rate parity checks, fee assumptions). Misconfigurations revert, preventing partially migrated or value-inconsistent states.

Cantina Managed: Acknowledged.

3.2 Gas Optimization

3.2.1 Inefficient storage array assignment

Severity: Gas Optimization

Context: StateSetterLib.sol#L146-L161

Description: When setting the deallocation order, it is built by pushing each strategy individually into the `$.deallocationOrder` storage array within a loop, leading to unnecessary repeated storage operations.

Recommendation: Assign the entire user provided order array to `$.deallocationOrder` at the end of the loop.

```diff
  function setDeallocationOrder(address[] calldata order) external {
      SVLib.ConcreteStandardVaultImplStorage storage $ = SVLib.fetch();

      delete $.deallocationOrder;

      uint256 orderLength = order.length;
      for (uint256 i = 0; i < orderLength; i++) {
          address strategy = order[i];
          require($.strategies.contains(strategy),
          ↪  IConcreteStandardVaultImpl.StrategyDoesNotExist());
          require(
              $.strategyData[strategy].status ==
              ↪  IConcreteStandardVaultImpl.StrategyStatus.Active,
              IConcreteStandardVaultImpl.StrategyIsHalted()
          );
-
-         $.deallocationOrder.push(strategy);
+     }
+
+     $.deallocationOrder = order;
  }
```

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.

3.2.2 Optimization opportunities

Severity: Gas Optimization

Context: ConcreteAsyncVaultImpl.sol#L105-L119, ConcreteStandardVaultImpl.sol#L695, ConcreteStandardVaultImpl.sol#L702, ConcreteStandardVaultImpl.sol#L903-L934

Description/Recommendations:

1. Avoid copying the entire deallocation order for small withdrawals.

   `_simulateWithdraw()` pulls the full `$.deallocationOrder` array into memory before it knows how many strategies it will touch. For small withdrawals the first strategy often satisfies the request, so iterating directly over storage (or copying chunks lazily) would avoid unnecessary memory allocations and lower gas costs on every call.

2. Redundant strategy membership guard.

   Both `_simulateWithdraw()` and `_executeWithdraw()` check

   ```
   $.strategies.contains(deallocationOrder[i])
   ```

   even though `removeStrategy()` prevents unregistered strategies from remaining in `deallocationOrder`. Since toggling a strategy to `Halted` never removes it from the set, the containment test can never fail during normal operations and only wastes gas. Consider deleting the extra membership check in both functions.

3. Management-fee share math can reuse cached ratios.

   `previewManagementFee()` derives fee shares by recomputing the asset-denominated fee and then converting it to shares using the adjusted total assets. Because the share mint is effectively `feeShares = f * totalSupply / (1 - f)` (where `f` is the time-delta fee fraction), the calculation could be expressed directly in terms of the fee rate and supply, saving reading asset values. The current logic is correct, so this is as a low-priority gas reduction.

4. Process-epoch flow can hardcode its precision scale.

   `ConcreteAsyncVaultImpl.processEpoch()` derives a share price by calling `decimals()` to determine how many assets correspond to one share-sized unit. The value only serves as a precision scaling factor, so it can be hardcoded to `10**18` (or another fixed precision) without depending on the vault's actual share decimals. Using a constant removes the repeated `decimals()` lookups, saves a storage read, and avoids passing the same number back through `AsyncVaultHelperLib.processEpoch()`.

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Inconsistencies in deposit/withdrawal limits definition

Severity: Informational

Context: ConcreteStandardVaultImpl.sol#L213-L214

Description: The system imposes deposit and withdrawal limits on the minimum and maximum amounts allowed. There is inconsistency in how these limits are defined. The `maxDeposit` limit applies globally, meaning the total deposits in the vault cannot exceed this amount. In contrast, the `minDeposit`, `minWithdraw`, and `maxWithdraw` limits are user-specific. For example, if `minDeposit` is set to a certain value, an individual user cannot deposit less than that amount, regardless of the total deposits in the vault.

Recommendation: To enhance clarity and consistency, standardize the terminology used for these limits. Consider renaming them to clearly distinguish between global and user-specific constraints (e.g., `globalMaxDeposit`, `userMinDeposit`, `userMinWithdraw`, `userMaxWithdraw`).

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.

3.3.2 Incorrect docs

Severity: Informational

Context: ConcreteStandardVaultImpl.sol#L673, Conversion.sol#L34-L38, UserDepositCapHook.sol#L104-L111

Description:

1. UserDepositCapHook.sol#L106: The NatSpec comment describing the parameters expected for the `preMint` function says the `sender` is the vault address, which is incorrect. It is the user that is paying to mint the vault shares.

2. ConcreteStandardVaultImpl.sol#L673: When describing what `_executeWithdraw` does, as a last step, the comment says it delegates to the parent contract for final ERC4626 withdrawal execution. This is not correct. The parent contract's `_withdraw` function never gets called. There is an overridden `_withdraw` method in the same contract that gets called.

3. Conversion.sol#L34-39: When converting from shares to assets, because the decimals offset is 1, both totalSupply and totalAssets are increased by 1. That is correct, but the comment says totalAssets is increased by `10  decimalsOffset`, which is not correct. `10  decimalsOffset` should be added to the totalSupply instead.

Recommendation: Correct the mistakes as recommended.

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.

3.3.3 User deposit limits can be bypassed

Severity: Informational

Context: UserDepositCapHook.sol#L148-L157

Description: The `UserDepositCapHook` is a pre-deposit hook that enforces per-user deposit limits across multiple vaults by checking the user's current balance plus the intended deposit amount against the owner set limit. However, without share transfer restrictions, users can bypass this by transferring shares to a new address, resetting their balance to zero before depositing more.

Recommendation: Implement transfer restrictions on vault shares for this limit to be meaningful.

Blueprint Finance: Acknowledged. We are aware of it, it is a design decision.

Cantina Managed: Acknowledged.

3.3.4 Strategy withdrawal reverts block vault deallocation

Severity: Informational

Context: ConcreteStandardVaultImpl.sol#L709-L715

Summary: A single reverting strategy halts the entire loop, leaving the vault unable to serve withdrawal requests that require other strategies even though sufficient liquidity may be available elsewhere.

Description: When redeeming user shares the vault iterates over `deallocationOrder` and queries each strategy for its available liquidity via `maxWithdraw()` before calling `onWithdraw()` to pull funds. Both calls are unguarded external interactions, so any revert propagates and aborts the whole `withdraw()/redeem()` transaction. Strategies are external contracts that may downgrade, be paused, or temporarily revert, so one misbehaving strategy can continue to block the vault even if remaining strategies could satisfy the withdrawal queue. This introduces a single point of failure that defeats the diversification intent of multiple strategies.

Impact Explanation: A reverting strategy prevents all pending withdrawals from completing until the issue is manually resolved, degrading availability but not directly losing funds.

Likelihood Explanation: Although unlikely, strategies can include their own guards and may revert under operator misconfiguration or downstream protocol issues, making this scenario plausible whenever any strategy experiences problems.

Recommendation: Consider wrapping the `maxWithdraw()` and `onWithdraw()` calls in `try/catch` blocks (or equivalent low-level calls) so the loop can skip failing strategies, optionally logging the failure and continuing with the remaining entries.

Blueprint Finance: We acknowledge this finding. The vault is intentionally designed to revert if any strategy in the deallocationOrder reverts during withdrawal. Strategies are assumed to be trusted and actively maintained components. A reverting strategy represents an abnormal condition that should be surfaced immediately rather than skipped silently. Using try/catch could hide failures and delay operator intervention.

As such, we accept the availability impact as a deliberate design choice: faulty strategies should be removed from the withdrawal queue rather than bypassed automatically.

Cantina Managed: Acknowledged.

3.3.5 Withdraw simulation ignores locked asset reserve

Severity: Informational

Context: ConcreteStandardVaultImpl.sol#L992

Description: `ConcreteStandardVaultImpl._simulateWithdraw()` treats the entire idle vault balance as withdrawable liquidity by querying `IERC20(asset()).balanceOf(address(this))`. The contract also supports strategy-specific locks via `_lockedAssets()`, but that value is never subtracted from the idle balance when previewing withdrawals. As a result, `_maxWithdraw()` and downstream ERC4626 previews may report more liquidity than can actually be sent because some assets are reserved for pending withdrawals, cross-chain claims, or other lockups enforced by `_lockedAssets()`.

Recommendation: Consider adjusting `_simulateWithdraw()` to subtract `_lockedAssets()` from the local balance before reporting available liquidity so that previews and max-withdraw calculations reflect the funds that are genuinely transferable.

Blueprint Finance: We acknowledge the finding. Although the vault cannot enter a state where `lockedAssets > idleBalance` due to the invariant enforced in `allocate()`, we agree that `_simulateWithdraw()` should still subtract `_lockedAssets()` for consistency with execution. We will apply this adjustment to keep ERC4626 previews accurate.

Cantina Managed: Acknowledged.

3.3.6 Implementation blocking relies on manual version lookup

Severity: Informational

Context: ConcreteFactory.sol#L81-L108

Description: `ConcreteFactory.blockImplementation()` and `setMigratable()` operate strictly on version numbers instead of the actual implementation addresses. Operators must first query each version $\rightarrow$ implementation mapping, confirm it matches the intended contract, and then call the management functions. When multiple upgrades are deployed or the mapping changes, this workflow becomes error-prone and increases the chance of blocking the wrong implementation or forgetting to mark a valid migration path.

Recommendation: Consider adding variants of these controls that accept implementation addresses directly (and internally resolve them to versions), or store the migration structure per implementation address instead of versions. This keeps the management interface aligned with how upgrades are referenced elsewhere and reduces manual bookkeeping during operations.

Blueprint Finance: Fixed in commit 2a94811b.

Cantina Managed: Fix verified.