Accountable Audit Report

Prepared by Cyfrin
Version 2.0

Lead Auditors

Immeas

Chinmay

Assisting Auditors

Alexzoid (Formal Verification)

October 16, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Accountable Loans provide fixed-term and open-term credit against pooled lender liquidity held in an "Accountable Async Vault." Lenders deposit the base asset to mint shares; a designated borrower draws and repays over time. In FixedTerm, interest is pre-scheduled per interval; in OpenTerm, a `_scaleFactor` (virtual share price) linearly accrues interest. Repayments (and, for FixedTerm, periodic `pay`) route protocol/establishment/performance fees to a `FeeManager`, return net interest to the vault, and update accounting. Withdrawals are served instantly when liquid, or queued and later processed via a withdrawal queue; in FixedTerm, deposits occur during a defined deposit window; in OpenTerm, deposits are allowed while the loan is "OngoingDynamic." Vault shares ultimately entitle lenders to principal and accrued interest pro-rata. Queuing requests can operate at current price or request-time price depending on vault mode.

4.1 Actors and Roles

- **1. Actors:**
  - **Lenders (LPs):** Deposit/withdraw the base asset; hold vault shares; request redemptions.
  - **Borrower:** Draws funds, supplies/repays, accepts/updates terms, and services interest.
  - **Investment Manager:** Configures loan terms, sets/updates borrower, governs upgrades/pauses.
  - **Fee Manager:** Receives protocol/establishment/performance fees.
  - **Safety Module / Provider:** Can cover defaults and move the loan into default-claims state.
- **2. Roles:**
  - **Manager (`onlyManager` / `onlyManagerOrSecurityAdmin`):** Sets terms, upgrades via UUPS, pauses/unpauses, defaults loans, covers defaults.
  - **Borrower (`onlyBorrower`):** Accepts terms, `borrow/supply/repay`, accepts/rejects updated terms.
  - **Safety Module (`onlySafetyModuleOrManager`):** Triggers default, covers default with assets.

- **Vault/Queue Operators (internal hooks):** Execute `onDeposit/onMint/onRequestRedeem`, process queues.

4.2 Key Components

- **AccountableAsync Vault + Withdrawal Queue:**
  - **Deposits/Minting:** LPs mint shares at strategy-provided price (FixedTerm: usually 1e18 during deposit period; OpenTerm: `_scaleFactor` outside deposit period).
  - **Redemptions:** `requestRedeem(shares)` queues withdrawals when liquidity is insufficient. Processing modes:
    - **CurrentPrice:** Queue pays at the share price when processed.
    - **RequestPrice:** Queue pays at the share price when requested.
  - **Processing:** Strategy or operator calls `processUpToShares` as liquidity becomes available; supports batched expiries/rollovers.
- **AccountableFixedTerm (strategy):**
  - **Lifecycle:** Manager sets fixed terms (rate, intervals, duration, deposit window). Borrower accepts, funds can be drawn up to capacity/min-cap constraints.
  - **Interest & Payments:** Interest accrues per interval; borrower calls `pay()` to settle interest/fees; `repay()` reduces principal; `prepay()` can settle early according to terms. Net interest is locked into the vault; fees go to `FeeManager`.
  - **Accounting:** Tracks `outstandingPrincipal`, `outstandingInterest`, `claimableInterest`, and `drawableFunds`. Lenders can claim interest (burn-for-yield model) or receive it via withdrawals depending on flow.
- **AccountableOpenTerm (strategy):**
  - **Lifecycle:** Manager sets terms; borrower accepts to enter `OngoingDynamic`. Deposits allowed while dynamic; borrower can `borrow/supply` and later `repay`.
  - **Accrual:** `_scaleFactor` increases linearly with time and delinquency penalties; share price equals `_scaleFactor` except during the initial deposit period or post-repay/default-claims, where it references vault's `assetShareRatio`.
  - **Delinquency & Penalties:** Loan tracks required liquidity vs. reserves to mark delinquency (with grace); penalties accrue while delinquent; safety module can default/cover.
  - **Repay Flow:** Assets are locked to the vault; queued withdrawals are processed; principal reduced; any configured fees are (intended to be) collected; loan moves to `Repaid` when principal is zero.
- **FeeManager:**
  - **Fees:** Calculates performance, establishment, and protocol split; `collect` is invoked by strategies during payments to route fees.
- **Upgrade & Guards:**
  - **UUPS Upgradeable:** Strategies and vaults are upgradeable by manager/admin.
  - **Pausable/Reentrancy Guards:** Critical state-changing flows are guarded; role checks enforce access.

4.3 Centralization risk

InvestmentManager control term setting, borrower designation, pausing, default/cover actions, and upgrades; the Borrower controls draw/repay cadence and can influence liquidity timing. Fee parameters reside in `FeeManager`. Compromise or misuse can impact interest accrual, redemptions, fee routing, and even default handling. We recommend multi-sig admins, timelocks for upgrades/term changes, strict key hygiene, and on-chain monitoring for role actions (term updates, accept/reject cycles, defaults) and queue processing events.

5 Audit Scope

```
src
   access
      AccessBase.sol
      Authorizable.sol
      Whitelistable.sol
   constants
      Errors.sol
   factory
      AsyncVaultFactory.sol
      FixedTermFactory.sol
      OpenTermFactory.sol
      RewardsFactory.sol
      StrategyFactoryBase.sol
   modules
      FeeManager.sol
      GlobalRegistry.sol
   rewards
      Rewards.sol
      RewardsDistributorMerkle.sol
      RewardsDistributorStrategy.sol
   strategies
      AccountableFixedTerm.sol
      AccountableOpenTerm.sol
      AccountableStrategy.sol
   vault
       AccountableAsyncRedeemVault.sol
       AccountableVault.sol
       queue
           AccountableWithdrawalQueue.sol
```

6 Executive Summary

Over the course of 13 days, the Cyfrin team conducted an audit on the Accountable smart contracts provided by Accountable. In this period, a total of 33 issues were found.

During the audit, four critical-severity findings were identified, all in the async withdrawal-queue processing. Three could leave the queue halted, permanently blocking withdrawals; the fourth enabled a share-price inflation attack when when withdrawal requests get partially fulfilled.

Two high-severity issues were also found: one involving the price selection when finalizing a redeem request, and another where an open-term loan cannot repay its final accrued interest once principal reaches zero.

In addition, we identified 12 medium-severity findings, plus several low and informational items.

The team added a commit `1d07a28` to change the owner of the strategy fractories and manager, which was also deemed to be safe.

Formal verification During the audit a formal verification suite was also developed by alexzoid, this was handed over to the protocol in PR#21, together with a formal verification report.

Post Audit Recommendations

Due to the significant number of Critical & High severity findings it is statistically likely that more serious vulnerabilities remain which could not be discovered during the 13-day audit window. Hence it is recommended that prior to deploying significant capital on-chain in a production environment, another audit be conducted during which no Critical or High severity findings should be found.

Summary

Project Name	Accountable
Repository	credit-vaults-internal
Commit	fc43546fe671...
Fix Commit	1ae7e2fb74a3...
Audit Timeline	Sep 22nd - Oct 8th, 2025
Methods	Manual Review, Formal Verification

Issues Found

Critical Risk	4
High Risk	2
Medium Risk	12
Low Risk	5
Informational	9
Gas Optimizations	1
Total Issues	33

Summary of Findings

[C-1] Cancelling redeem requests permanently blocks the withdrawal queue	Resolved
[C-2] `AccountableAsyncRedeemVault::fulfillCancelRedeemRequest` can de-sync request data causing permanent DOS for queue processing	Resolved
[C-3] Critical DOS in queue processing if async cancellations are allowed	Resolved
[C-4] Partial redemptions can be used to steal assets	Resolved
[H-1] `AccountableAsyncRedeemVault::fulfillRedeemRequest` ignores processingMode and directly uses currentPrice for finalizing a redeem request	Resolved
[H-2] `AccountableOpenTerm` loan interest cannot be repaid once principal hits zero	Resolved
[M-01] Complete bypass of transfer restrictions on vault share token is possible	Resolved
[M-02] `transferWhitelist` checks are missing in `AccountableVault::_checkTransfer`	Resolved
[M-03] `AccountableAsyncRedeemVault` allows deposits for non-whitelisted or non-KYCed addresses	Resolved
[M-04] InvestmentManager can use `AccountableFixedTerm::coverDefault` to misuse token approvals from anyone	Resolved
[M-05] Manual/Instant `fulfillRedeemRequest` doesn't reserve liquidity	Resolved
[M-06] `AccountableFixedTerm::claimInterest` unpredictable due to share burn mechanics	Resolved

[M-07] Fees never deducted in `AccountableOpenTerm` loan	Resolved
[M-08] Borrower in OpenTerm loan can stay delinquent effectively forever	Acknowledged
[M-09] Withdrawal queue `RequestPrice` can be front run in case of defaults	Resolved
[M-10] Auto-draw on `AccountableFixedTerm::pay` lets third parties force unwanted borrowing	Resolved
[M-11] Frequent `AccountableOpenTerm::accrueInterest` calls reduce interest accrual	Resolved
[M-12] Invalid `maxWithdraw()` check in `withdraw()`	Resolved
[L-1] Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collisions	Resolved
[L-2] Missing controller validation in `AccountableAsyncRedeemVault::requestRedeem` allows zero address state	Resolved
[L-3] Reserved assets could be extracted from the Vault	Resolved
[L-4] `Authorizable::_verify` should use EIP-712 typed structured data hashing	Resolved
[L-5] Deployment script requires unencrypted private key	Resolved
[I-1] Prevent accidental ownership and admin renouncement	Resolved
[I-2] Consider consistently use `Ownable2Step`	Resolved
[I-3] Consider enforcing a minimum deposit amount	Resolved
[I-4] Violations of ERC7540 specs	Acknowledged
[I-5] Incorrect event emission is possible in `AccountableAsyncRedeemVault::cancelRedeemRequest` flows	Resolved
[I-6] ERC20 zero amount transfer rejection	Resolved
[I-7] `nonReentrant` is not the first modifier	Resolved
[I-8] Unused errors	Resolved
[I-9] State changes without events	Resolved
[G-1] Storage read optimizations	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Cancelling redeem requests permanently blocks the withdrawal queue

Description: `AccountableWithdrawalQueue` can deadlock at the head if the current head entry (`_queue.nextRequestId`) is fully removed (e.g., by a cancel that zeroes `shares` and clears `controller`) without advancing `nextRequestId`.

In `AccountableWithdrawalQueue::_processUpToShares` and `AccountableWithdrawalQueue::_processUpToRequestId`, the loop checks `if (shares_ == 0) break;` before incrementing `nextRequestId`:

```
(uint256 shares_, uint256 assets_, bool processed_) =
    _processRequest(request_, liquidity, maxShares_, precision_);

if (shares_ == 0) break;
```

When the head is an empty entry (`controller == address(0)`), `AccountableWithdrawalQueue::_processRequest` returns `(0, 0, true)`, `shares_ == 0`, the loop breaks:

```
if (request.controller == address(0)) return (0, 0, true);
```

The head never advances, so every subsequent call to process or preview gets stuck on the same empty head forever.

This can be triggered by any user whose request is currently at the head by canceling any dust amount (even 1 wei) such that their head entry is fully deleted at the time of processing (e.g., instant cancel-fulfillment) in `AccountableWithdrawalQueue::_delete`:

```
/// @dev Deletes a withdrawal request and its controller from the queue
function _delete(address controller, uint128 requestId) private {
    delete _queue.requests[requestId];
    delete _requestIds[controller];
}
```

Once the head becomes an empty slot and the pointer doesn't move, the entire queue is bricked.

Impact: Queue is permanently stuck and no subsequent user will be able to withdraw.

Proof of Concept: Add the following test to `test/vault/AccountableWithdrawalQueue.t.sol`:

```
function testHeadDeletionDeadlocksQueue() public {
    // Setup: deposits are instant, redemptions are queued, cancel is instantly fulfilled
    strategy.setInstantFulfillDeposit(true);
    strategy.setInstantFulfillRedeem(false);
    strategy.setInstantFulfillCancelRedeem(true);

    // Seed vault with liquidity and create first (head) request by Alice
    // This helper deposits for Alice and Bob at 1e36 price.
    _setupInitialDeposits(1e36, DEPOSIT_AMOUNT);

    // 1) Alice creates a redeem request -> head of queue (requestId = 1)
    uint256 aliceSharesToQueue = 1;
    vm.prank(alice);
    uint256 headId = vault.requestRedeem(aliceSharesToQueue, alice, alice);
    assertEq(headId, 1, "first request should be head (id = 1)");

    // 2) Alice cancels; cancel is fulfilled instantly by the strategy.
    //    This fully removes the head request entry (controller becomes address(0)),
    //    but _queue.nextRequestId is NOT advanced by the implementation.
    vm.prank(alice);
    vault.cancelRedeemRequest(headId, alice);
```

```
    // Sanity: queue indices should still point at the deleted head
    (uint128 nextRequestId, uint128 lastRequestId) = vault.queue();
    assertEq(nextRequestId, 1, "nextRequestId remains stuck at deleted head");
    assertGe(lastRequestId, 1, "there is at least one request in the queue history");

    // 3) Charlie makes a NEW redeem request -> tail (requestId = 2).
    //    This request is perfectly processable with existing liquidity.
    token.mint(charlie, 1000e6);
    vm.prank(charlie);
    token.approve(address(vault), 1000e6);
    vm.prank(charlie);
    vault.deposit(1000e6, charlie);

    uint256 charlieShares = vault.balanceOf(charlie) / 2;
    vm.prank(charlie);
    uint256 tailId = vault.requestRedeem(charlieShares, charlie, charlie);
    assertEq(tailId, 2, "second request should be tail (id = 2)");

    // Check queue bounds reflect head(=1, deleted) and tail(=2, valid)
    (nextRequestId, lastRequestId) = vault.queue();
    assertEq(nextRequestId, 1, "still pointing at deleted head");
    assertEq(lastRequestId, 2, "tail id should be 2");

    // 4) Attempt to process. BUG: _processUpToShares reads head slot (controller==0),
    //    inner _processRequest returns (0,0,true), outer loop sees shares_==0 and BREAKS
    //    BEFORE ++nextRequestId, so NOTHING gets processed and the queue is permanently stuck.
    uint256 assetsBefore = vault.totalAssets();
    uint256 used = vault.processUpToShares(type(uint256).max);
    assertEq(used, 0, "deadlock: processing does nothing while a valid tail exists");

    (uint256 _shares, uint256 _assets) = vault.processUpToRequestId(2);
    assertEq(_shares, 0, "deadlock: processing does nothing while a valid tail exists");
    assertEq(_assets, 0, "deadlock: processing does nothing while a valid tail exists");

    // 5) Verify tail wasn't progressed at all
    assertEq(vault.claimableRedeemRequest(0, charlie), 0, "tail remains unclaimable");
    assertEq(vault.pendingRedeemRequest(0, charlie), charlieShares, "tail remains fully pending");
    assertEq(vault.totalAssets(), assetsBefore, "no reserves changed due to deadlock");
    (nextRequestId, lastRequestId) = vault.queue();
    assertEq(nextRequestId, 1, "nextRequestId is still stuck at deleted head");
}
```

Recommended Mitigation: Consider incrementing the counter if it's processed, and `continue` instead of break:

```
if (shares_ == 0) {
    if (processed_) {
        ++nextRequestId;
        continue;
    }
    break;
}
```

Accountable: Fixed in commits 2df3becf and b432631

Cyfrin: Verified. The counter is now incremented if the request was processed even if shares were 0.

7.1.2 `AccountableAsyncRedeemVault::fulfillCancelRedeemRequest` can de-sync request data causing permanent DOS for queue processing

Description: `fulfillCancelRedeemRequest()` function first finalises the cancellation of the redeeem request with input `requestID`, and then calls `_reduce()` to update the request state and `totalQueuedShares`.

```
    function fulfillCancelRedeemRequest(address controller) public onlyOperatorOrStrategy {
        _fulfillCancelRedeemRequest(_requestIds[controller], controller);
        _reduce(controller, _vaultStates[controller].pendingRedeemRequest);
    }
```

The problem here is that it is using current value of `_vaultStates[controller].pendingRedeemRequest` in the `_reduce()` call, but it has been set to zero in `_fulfillCancelRedeemRequest()`.

This means `_reduce()` here will always be called with zero shares, and it does not revert when shares input is zero. But it corrupts the request struct and `totalQueuedShares` value.

The request will still exist with actual shares values, and create problems in usual batch processing of the queue.

One example of the resulting impact is this :

1. User X places a redeem request for 100 shares
2. User X cancels this redeem request
3. His request is not fulfilled instantly (this depends on the strategy)
4. Operator calls `fulfillCancelRedeemRequest()` to process this cancellation.
5. The call goes through properly. As a result [state.pendingRedeemRequest = 0] but the request state still has request.shares == 100 and other values. Also, the `_queue.nextRequestID` remains unchanged.
6. Now when batch processing proceeds via `processUpToShares()`, it is guaranteed that User X's requestID will also be processed (it is still in the queue from nextRequestID to lastRequestID) and when that happens, it will suffer a revert in `_processUptoShares() => _fulfillRedeemRequest()` because `state.pendingRedeemRequest` was set to == 0 in step 5.

```
    function _fulfillRedeemRequest(uint128 requestId, address controller, uint256 shares, uint256 price)
        internal
        override
    {
        VaultState storage state = _vaultStates[controller];
        if (state.pendingRedeemRequest == 0) revert NoRedeemRequest();
        if (state.pendingRedeemRequest < shares) revert InsufficientAmount();
        if (state.pendingCancelRedeemRequest) revert RedeemRequestWasCancelled();
```

Impact: If this function is ever called, there will be a permanent de-sync between the values stored as per requestID data and the vaultState of the controller, which will interfere with queue processing in different ways.

The example showcased here is a critical DOS blocking queue processing permanently. This will happen for strategies that offer async cancellation processing, but since vault is expected to be compatible with this behavior, fixing this is critical.

Recommended Mitigation:

```
    function fulfillCancelRedeemRequest(address controller) public onlyOperatorOrStrategy {

+++        uint256 pendingShares = state.pendingRedeemRequest;
               _fulfillCancelRedeemRequest(_requestIds[controller], controller);
---          _reduce(controller, _vaultStates[controller].pendingRedeemRequest);
+++        _reduce(controller, pendingShares);
    }
```

Accountable: Fixed in commit 84946dd

Cyfrin: Verified. `pendingShares` now cached before fulfill and then passed as argument to `_reduce`.

7.1.3 Critical DOS in queue processing if async cancellations are allowed

Description: The `cancelRedeemRequest()` function can be used to DOS the queue processing (ie. `processUpToShares()` and `processUpToRequestID()` can be made to revert).

This is the attack path :

- `cancelRedeemRequest()` marks `state.pendingCancelRedeemRequest = true`;
- Assume that this cancellation is not instantly fulfilled, as the associated strategy may support async cancellations

```
function cancelRedeemRequest(uint256 requestId, address controller) public onlyAuth {
    _checkController(controller);
    VaultState storage state = _vaultStates[controller];
    if (state.pendingRedeemRequest == 0) revert NoPendingRedeemRequest();
    if (state.pendingCancelRedeemRequest) revert CancelRedeemRequestPending();

    state.pendingCancelRedeemRequest = true;

    bool canCancel = strategy.onCancelRedeemRequest(address(this), controller); // @audit strategy
    ↪  can choose to return false here, thus mandating async cancellations.
    if (canCancel) {
        uint256 pendingShares = state.pendingRedeemRequest;

        _fulfillCancelRedeemRequest(uint128(requestId), controller);
        _reduce(controller, pendingShares);
    }
    emit CancelRedeemRequest(controller, requestId, msg.sender);
}
```

- At this step, it also skips "reducing" the shares in request state, as _reduce() will only be called when cancellation is fulfilled via `fulfillCancelRedeemRequest()`
- Later when `processUpToShares()` is called, `_processRequest()` returns normal request data (does not return "zero values" as request.shares was not reduced in the cancel logic) => so it doesn't break the loop or continue with nextRequestID
- It goes on to call `_fulfillRedeemRequest()`, where it reverts due to pendingCancelRedeemRequest = true

```
function _fulfillRedeemRequest(uint128 requestId, address controller, uint256 shares, uint256 price)
    internal
    override
{
    VaultState storage state = _vaultStates[controller];
    if (state.pendingRedeemRequest == 0) revert NoRedeemRequest();
    if (state.pendingRedeemRequest < shares) revert InsufficientAmount();
    if (state.pendingCancelRedeemRequest) revert RedeemRequestWasCancelled();  // @audit
```

This means even a single async cancellation (that is pending for processing) can DOS queue processing.

Impact: Queue processing can be repeatedly DOS'ed under normal operations as well as by an attacker frontrunning a process call, in case the strategy contract allows async cancellations.

Recommended Mitigation: Consider removing async cancellations' support from the system, which prevents this kind of attacks.

Accountable: Fixed in commit 2eeb273

Cyfrin: Verified. Async cancelation of redeem requests now removed.

7.1.4 Partial redemptions can be used to steal assets

Description: The request state is not handled properly when redeem requests are filled partially, leading to an inflated redemption price for the remaining part of the request.

- When a new redemption is pushed onto an existing requestID, then the average redemption price is calculated using the updated `totalValue` and updated `request.shares`. This is then stored as the `request.sharePrice` (used for calculating assets owed for those shares).

```
        } else { // if controller had an existing active requestID
            requestId = requestId_;

            WithdrawalRequest storage request = _queue.requests[requestId_];

            request.shares += shares;

            if (processingMode == ProcessingMode.RequestPrice) {
                request.totalValue += shares.mulDiv(sharePrice, _precision);
                request.sharePrice = request.totalValue.mulDiv(_precision, request.shares); // the
                ↪  average sharePrice is being calculated here.
            } // the whole request will have a single price, averaged recursively as new redeem requests
            ↪  come up.

        totalQueuedShares += shares;
    }
```

- This works fine when request is fulfilled completely or cancelled completely as in those cases request data gets wiped out. But the problem is that when such a request is filled partially, this totalValue is never decreased while request.shares is decreased.

```
    function _reduce(address controller, uint256 shares) internal returns (uint256 remainingShares) {
        uint128 requestId = _requestIds[controller];
        if (requestId == 0) revert NoQueueRequest();

        uint256 currentShares = _queue.requests[requestId].shares;
        if (shares > currentShares || currentShares == 0) revert InsufficientShares();

        remainingShares = currentShares - shares;
        totalQueuedShares -= shares;

        if (remainingShares == 0) {
            _delete(controller, requestId);
        } else {
            _queue.requests[requestId].shares = remainingShares;
        } // @audit the totalValue is not updated here.
    }
```

This is the attack path :

- User places a redeem request for 100 shares at a time when sharePrice == 2. So the request data stored is => {request.totalValue = 200, request.sharePrice = 2, request.shares = 100}.
- This request gets fulfilled partially ie. 50 shares. Resultant state => {request.totalValue = 200, request.sharePrice = 2, request.shares = 50}. User got 100 assets.
- User places another redeem request with 100 shares for the same controller address, thus the same requestID data will be modified. The new sharePrice will be calculated using an inflated "request.totalValue" and a normal request.shares. As per the calculation, the resultant state => {request.totalValue = 400, request.shares = 150, and request.sharePrice = 2.66}
- Assume this request gets filled completely. User now gets 400 assets.

User got a total of 500 assets for redeeming 200 shares, even though the sharePrice was only 2. This is because the calculation uses an inflated value of request.totalValue to calculate the redemption price.

- This request.sharePrice is used when calculating assets owed to the controller in `_fulfillRedeemRequest()` flow

This means an inflated amount of assets will be added to the VaultState.maxWithdraw => allowing controller to claim more assets than they deserved if actual sharePrice was used.

Note : Partial redemption is possible when `fulfillRedeemRequest()` is called with a portion of the request's shares, and also possible when `processUptoShares()` is used and it hits a block with maxShares/ liquidityShares (such that a particular request is not processed completely.

Impact: An attacker can steal assets easily if their redeem request was fulfilled partially, in case the vault is configured with a processingMode == RequestPrice.

This issue exists only when processingMode == RequestPrice, as only then the request.sharePrice value is used for calculating assets owed.

Recommended Mitigation: Consider removing the processingMode logic entirely to simplify the system, or decrease redeemed assets from `request.totalValue` as part of the `_reduce()` function.

Accountable: Fixed in commit `4e5eef5`

Cyfrin: Verified. `processingMode` removed as well as `totalValue`.

7.2 High Risk

7.2.1 `AccountableAsyncRedeemVault::fulfillRedeemRequest` ignores processingMode and directly uses currentPrice for finalizing a redeem request

Description: When a redeem request is placed using `requestRedeem` function, it pushes a new request struct into the withdrawal queue. If the processingMode of the vault is configured to be `== RequestPrice`, the current `sharePrice` at that time is stored as the "request.sharePrice" for later use when the request will be processed.

All functions in the `AccountableWithdrawalQueue` honour this price and the assets user receives depends on this stored sharePrice (in case processingMode == RequestPrice).

But there is one function in `AccountableAsyncRedeemVault` that ignores the processing mode and uses the current `sharePrice`.

```
    function fulfillRedeemRequest(address controller, uint256 shares) public onlyOperatorOrStrategy {
        _fulfillRedeemRequest(_requestIds[controller], controller, shares, sharePrice());
        _reduce(controller, shares);
    }
```

The `sharePrice` here fetches the current price of the shares, but if the `sharePrice` changed since the request time, it can be unfavourable to the user as he could get lesser amount of assets just because of the delay in processing, and that should not happen when the `processingMode == RequestPrice`.

Impact: For a vault configured with `processingMode == RequestPrice`, the `fulfillRedeemRequest` functions breaks the guarantee that the price stored at time of placing the redeem request would be used for calculating the assets user gets in return, which might be unfavourable if the sharePrice decreased due to any reason.

Recommended Mitigation:

```
    function fulfillRedeemRequest(address controller, uint256 shares) public onlyOperatorOrStrategy {
+++         uint256 price;
+++         if (processingMode == ProcessingMode.CurrentPrice)
+++             price = sharePrice();
+++        }
+++         else {
+++             uint128 requestId = _requestIds[controller];
+++             price = _queue.requests[requestId].sharePrice;
+++       }

              _fulfillRedeemRequest(_requestIds[controller], controller, shares, price);
              _reduce(controller, shares);
          }
```

Accountable: Not applicable due to `processingMode` being removed in commit 4e5eef5

7.2.2 `AccountableOpenTerm` loan interest cannot be repaid once principal hits zero

Description: In `AccountableOpenTerm`, interest accrues virtually via `_scaleFactor`, but there is no mechanism to pay/realize that interest. The only funding paths are `borrow()`, `supply()`, and `repay()`. Both `supply()` and `repay()` first service withdrawals and then reduce `_loan.outstandingPrincipal`. When principal reaches zero, `repay()` sets `loanState = Repaid`; in `Repaid`, `_requireLoanOngoing()` blocks further `supply()/repay()`, and `_sharePrice()` switches to `assetShareRatio()` (ignoring accrued `_scaleFactor`). As a result, any accrued interest becomes unpayable and is never delivered to LPs (or fee recipients).

Impact: If the borrower repays the principa after time has passed, the loan flips to `Repaid` and all accrued interest is effectively forgiven. LPs receive principal back with zero interest. A borrower can always avoid interest by repaying principal before realizing it. This can happen intentionally by a malicious borrower or even unintentionally since the payments decrease principal first. Hence all interest needs to be repaid with the last payment.

Proof of Concept: Add the following test to `AccountableOpenTerm.t.sol`:

```
function test_openTerm_repay_principal_only_setsRepaid_no_interest_paid() public {
```

```
vm.warp(1739893670);

// Setup borrower & terms
vm.prank(manager);
usdcLoan.setPendingBorrower(borrower);
vm.prank(borrower);
usdcLoan.acceptBorrowerRole();

LoanTerms memory terms = LoanTerms({
    minDeposit: 0,
    minRedeem: 0,
    maxCapacity: USDC_AMOUNT,
    minCapacity: USDC_AMOUNT / 2,
    interestRate: 150_000,          // 15% APR so scale factor grows visibly
    interestInterval: 30 days,
    duration: 0,
    depositPeriod: 2 days,
    acceptGracePeriod: 0,
    lateInterestGracePeriod: 0,
    lateInterestPenalty: 0,
    withdrawalPeriod: 0
});
vm.prank(manager);
usdcLoan.setTerms(terms);
vm.prank(borrower);
usdcLoan.acceptTerms();

// Single LP deposits during deposit period → 1:1 shares at PRECISION
vm.prank(alice);
usdcVault.deposit(USDC_AMOUNT, alice, alice);
assertEq(usdcVault.totalAssets(), USDC_AMOUNT, "vault funded");

// Borrower draws full principal → vault drained
vm.prank(borrower);
usdcLoan.borrow(USDC_AMOUNT);
assertEq(usdcVault.totalAssets(), 0, "all assets borrowed");
assertEq(usdcLoan.loan().outstandingPrincipal, USDC_AMOUNT, "principal outstanding");

// Time passes → interest accrues virtually (scale factor > PRECISION)
vm.warp(block.timestamp + 180 days);
uint256 sfBefore = usdcLoan.accrueInterest();
assertGt(sfBefore, 1e36, "scale factor increased (virtual interest)");

// Borrower repays EXACTLY principal (no extra for interest)
usdc.mint(borrower, USDC_AMOUNT);
vm.startPrank(borrower);
usdc.approve(address(usdcVault), type(uint256).max);
usdcLoan.repay(USDC_AMOUNT);
vm.stopPrank();

// Loan marked repaid even though totalAssets < totalShareValue at sfBefore
assertEq(uint8(usdcLoan.loanState()), uint8(LoanState.Repaid), "loan flipped to Repaid");

// After Repaid, share price uses assetShareRatio (actual assets), not the higher scale factor.
// With one LP and totalAssets == totalSupply, ratio == PRECISION → no interest realized.
uint256 spAfter = usdcLoan.sharePrice(address(usdcVault));
assertEq(spAfter, 1e36, "share price fell back to assetShareRatio (no interest paid)");

// Sanity: vault now only holds repaid principal
assertEq(usdcVault.totalAssets(), USDC_AMOUNT, "vault holds only principal after repay");
assertEq(usdcVault.totalSupply(), USDC_AMOUNT, "shares unchanged");

// Now borrower cannot "pay the interest" anymore
```

```
    vm.prank(borrower);
    vm.expectRevert(); // blocked by _requireLoanOngoing()
    usdcLoan.supply(1e6);
}
```

Recommended Mitigation: Consider modeling borrower liability in debt shares instead of tracking principal/interest separately.

On `borrow(assets)`, after `accrue()`, mint `debtShares = ceil(assets * PRECISION / price)` where `price = scaleFactor`. Debt then equals `debtShares * price / PRECISION`. Interest accrual only moves `price`, not shares.

On `repay(assets)`, after `accrue()`, burn `sharesToBurn = floor(assets * PRECISION / price)` (capped to balance). When `debtShares == 0`, the loan is repaid.

This guarantees interest can always be repaid at the current price, prevents the "principal hits zero" dead-end, and supports partial/frequent repayments cleanly. If protocol/establishment fees apply, take them on each accrual/settle step before any excess refunds to keep fee accounting correct.

Accountable: Fixed in commits `fce6961` and `8e53eba`

Cyfrin: Verified. Debt is not tracked using shares.

7.3 Medium Risk

7.3.1 Complete bypass of transfer restrictions on vault share token is possible

Description: In `AccountableVault.sol` (which is inherited by the `AccountableAsyncRedeemVault`, we have certain transfer restrictions (KYC, if from address is subject to a throttle timestamp), applied in `_checkTransfer()` function.

These restrictions are applied on `transfer()`/ `transferFrom()` function (inherited from ERC20) when share holders try to move their holdings.

These restrictions do not apply when the internal `_transfer()` function is used, which is fine for most cases as these share tokens will be moved only for deposits and redeems.

But there is one case where user can use the `cancelRedeemRequest()` feature to bypass all these restrictions completely, and move share tokens to a different address.

This is how it can be done :

- Assume controller has a deposit in the vault
- Controller places a redeem request
- Controller immediately cancels the redeem request
- Controller calls `claimCancelRedeemRequest()` where share tokens are transferred to a "receiver" address

```
function claimCancelRedeemRequest(uint256 requestId, address receiver, address controller)
     public
     onlyAuth
     returns (uint256 shares)
 {
     _checkController(controller);
     VaultState storage state = _vaultStates[controller];
     shares = state.claimableCancelRedeemRequest;
     if (shares == 0) revert ZeroAmount();

     strategy.onClaimCancelRedeemRequest(address(this), controller);

     state.claimableCancelRedeemRequest = 0;

     _transfer(address(this), receiver, shares); // @audit bypasses all transfer restrictions.

     emit CancelRedeemClaim(receiver, controller, requestId, msg.sender, shares);
 }
```

For this transfer step, the internal `_transfer()` function is used which skips all transfer restrictions applicable as per AccountableVault logic.

Impact: This "receiver" address input while calling `claimCancelRedeemRequest()` is the controller's choice and there are no checks on it as `_checkTransfer()` gets bypassed. This allows to transfer shares even if "to" address is not KYC-ed or transfers originating at "from" address had to work with a cooldown time.

This way controller is able to move their vault shares to a random receiver address, bypassing the transfer restrictions.

Recommended Mitigation: In `claimCancelRedeemRequest()`, remove the receiver address logic and just transfer the cancelled shares back to the controller address. This solves the issue as controller is already expected to be KYC-ed, and there will be no need for a cooldown check in that case as shares are going back to the original holder.

Accountable: Fixed in commit 2eeb273

Cyfrin: Verified. `reciever` now checked against KYC.

7.3.2 `transferWhitelist` checks are missing in `AccountableVault::_checkTransfer`

Description: AccountableVault.sol employs a "transferWhitelist" feature to help select addresses that should be allowed to transfer vault shares, overriding the other restrictions checked in `_checkTransfer()`.

Both `transfer()` and `transferFrom()` functions internally call `_checkTransfer()`, but the "transferWhitelist" check is missing in all transfer flows.

Impact: The transferWhitelist feature does not work, so it does not make a difference if an address was whitelisted or not.

```
/// @notice Mapping of addresses that can override transfer restrictions
mapping(address => bool) public transferWhitelist;
```

The comment above says "Mapping of addresses that can override transfer restrictions" which does not hold true as transferWhitelist is never being checked.

A method to call vault.setTransferWhitelist() is also missing in both the current strategy contracts, so when fixing keep note of it.

Recommended Mitigation:

```
    function _checkTransfer(uint256 amount, address from, address to) private {

+++      if(transferWhitelist[from] && transferWhitelist[to]) return;

        if (amount == 0) revert ZeroAmount();
        if (!transferableShares) revert SharesNotTransferable();
        if (!isVerified(to, msg.data)) revert Unauthorized();
        if (throttledTransfers[from] > block.timestamp) revert TransferCooldown();
    }
```

Also consider adding a method to the AccountableFixedTerm and AccountableOpenTerm strategy contracts (one that calls vault.setTransferWhitelist()) if it is required in context of that strategy.

Accountable: Whitelist removed in commit 6a81e38

Cyfrin: Verified. Whitelist removed.

7.3.3 `AccountableAsyncRedeemVault` allows deposits for non-whitelisted or non-KYCed addresses

Description: Almost all functions in `AccountableAsyncRedeemVault` use an `onlyAuth()` modifier to verify that the caller is KYC-ed or Whitelisted (according to the vault's own policy).

This logic can be seen in `isVerified()` function in AccessBase.sol

Here is the `AccountableAsyncRedeemVault::onlyAuth` modifier :

```
modifier onlyAuth() {
    if (!isVerified(msg.sender, msg.data)) revert Unauthorized();
    _;
}
```

This passes `msg.sender` as the "Account" address to be verified, but these checks are not working.

If we look at the `deposit()` function here, `msg.sender` is not the actual account address, for whom the deposit will be done, instead the "receiver" address here is the actual account. The "Receiver" address receives the shares but it is not verified that they are whitelisted/ KYC-ed.

```
function deposit(uint256 assets, address receiver, address controller) public onlyAuth returns
↪   (uint256 shares) {
    _checkController(controller);
    if (assets == 0) revert ZeroAmount();
    if (assets > maxDeposit(controller)) revert ExceedsMaxDeposit();
```

```
    uint256 price = strategy.onDeposit(address(this), assets, receiver, controller);
    shares = _convertToShares(assets, price, Math.Rounding.Floor);

    _mint(receiver, shares);
    _deposit(controller, assets);
```

This means that a KYC'ed user can call `deposit()` and mint new share tokens for random "receiver" addresses (who have set the KYC'ed user as their operator using `setOperator()` and for the input params `controller == receiver` can be used). This "receiver" can then take part in the vault by holding vault shares, redeeming them via the operator etc.

Impact: The KYC/ Whitelist configuration does not prevent KYC'ed addresses from minting shares to non-KYCed addresses.

Similar problems might exist in the access control for other methods in the vault, the reason being `onlyAuth()` only checks the msg.sender and not the other address holding the position.

Recommended Mitigation: Consider documenting what is the intended permissions granted to a KYC-ed/ Whitelisted user. If they should not be allowed to open positions for other non KYC-ed addresses, then the auth checks need to be done for actual receiver/ controller addresses.

Accountable: Fixed in commits `c804a31` and `2eeb273`

Cyfrin: Verified. Both `reciever` and `controller` are verified to be KYC'd throughout the calls.

7.3.4 InvestmentManager can use `AccountableFixedTerm::coverDefault` to misuse token approvals from anyone

Description: `AccountableFixedTerm::coverDefault` allows InvestmentManager of the loan to add additional assets to the system.

```
function coverDefault(uint256 assets, address provider) external onlySafetyModuleOrManager
↪   whenNotPaused {
    _requireLoanInDefault();

    loanState = LoanState.InDefaultClaims;

    IAccountableVault(vault).lockAssets(assets, provider);

    emit DefaultCovered(safetyModule, provider, assets);
}
```

And `lockAssets()` pulls assets from the input "provider" address, transferring them to the vault.

This means any user address who had asset token balance, and approved the vault contract (potential pending approvals from the past) is at risk of losing their funds here.

The Manager can pull funds from a random provider address without any permissions, and the "provider" would lose his approved funds without getting anything in return.

Impact: Any pending asset approvals from user => vault contract, can be misused to cover loan default.

The same problem also exists in AccountableOpenTerm.

Recommended Mitigation: Consider removing the "provider" address logic from `coverDefault()`, and simply pull assets from `msg.sender`.

Accountable: Fixed in commit `014d7fb`

Cyfrin: Verified. `provider` is removed.

7.3.5 Manual/Instant `fulfillRedeemRequest` doesn't reserve liquidity

Description: `AccountableAsyncRedeemVault` account for reserved liquidity only when processing the queue through (`AccountableWithdrawalQueue::processUpToShares` / `AccountableWithdrawalQueue::processUpToRequestId`). However, the manual fulfillment paths (`fulfillRedeemRequest`) and the instant branch of `requestRedeem` mark shares as claimable without increasing `reservedLiquidity`.

When these paths are mixed, multiple fulfillments can each pass the "liquidity" check independently (because nothing was reserved by earlier fulfills), producing a state where:

```
sum(claimable assets across users)  >  vault.totalAssets() - reservedLiquidity
```

Potentially causing claimable assets to be larger than the available liquidity.

Impact: The vault can end up with more claimable redemptions than available assets, causing later withdrawals to revert (depending on integration logic), and creating fairness and accounting issues between users.

Proof of Concept: Add the following test to `AccountableWithdrawalQueue.t.sol`:

```
function test_manualFulfill_vsQueuedFulfill_mismatch() public {
    // Setup: price = 1e36, deposits for Alice & Bob
    _setupInitialDeposits(1e36, DEPOSIT_AMOUNT);

    uint256 aliceHalf = vault.balanceOf(alice) / 2;
    uint256 bobHalf   = vault.balanceOf(bob)   / 2;

    // === (A) Queue Bob first and reserve via processor ===
    vm.prank(bob);
    uint256 bobReqId = vault.requestRedeem(bobHalf, bob, bob);
    assertEq(bobReqId, 1, "Bob should be the head of the queue");

    // Processor path reserves liquidity for Bob
    uint256 price = strategy.sharePrice(address(vault)); // 1e36
    uint256 expectedBobAssets = (bobHalf * price) / 1e36;
    uint256 used = vault.processUpToShares(bobHalf);
    assertEq(used, expectedBobAssets, "queued fulfill reserves exact assets for Bob");

    // Sanity: reservedLiquidity == Bob's claimable assets
    uint256 reservedBefore = vault.reservedLiquidity();
    assertEq(reservedBefore, expectedBobAssets, "only Bob's queued path bumped reservedLiquidity");

    // === (B) Now manually fulfill Alice (no reservation bump) ===
    vm.prank(alice);
    uint256 aliceReqId = vault.requestRedeem(aliceHalf, alice, alice);
    assertEq(aliceReqId, 2, "Alice should be behind Bob in the queue");

    // Manual fulfill creates claimables but doesn't increase reservedLiquidity
    strategy.fulfillRedeemRequest(0, address(vault), alice, aliceHalf);

    // Compute claimables in assets
    uint256 aliceClaimableShares = vault.claimableRedeemRequest(0, alice);
    uint256 bobClaimableShares   = vault.claimableRedeemRequest(0, bob);
    assertEq(aliceClaimableShares, aliceHalf, "Alice claimable shares set by manual fulfill");
    assertEq(bobClaimableShares,   bobHalf,   "Bob claimable shares set by queued processor");

    uint256 aliceClaimableAssets = (aliceClaimableShares * price) / 1e36;
    uint256 bobClaimableAssets   = (bobClaimableShares   * price) / 1e36;
    uint256 totalClaimables      = aliceClaimableAssets + bobClaimableAssets;

    // Mismatch: claimables exceed reservedLiquidity because Alice's path didn't reserve
    assertGt(totalClaimables, reservedBefore, "claimables > reservedLiquidity (oversubscription)");

    // === (C) Bob withdraws his reserved claim -> consumes all reservation ===
    uint256 bobMax = vault.maxWithdraw(bob);
```

```
    assertEq(bobMax, bobClaimableAssets, "Bob can withdraw exactly his reserved amount");

    uint256 vaultAssetsBefore = vault.totalAssets();
    vm.prank(bob);
    vault.withdraw(bobMax, bob, bob);

    // After paying Bob, reservation is zero, but Alice still has claimables (unreserved)
    uint256 reservedAfter = vault.reservedLiquidity();
    assertEq(reservedAfter, 0, "all reserved liquidity consumed by Bob's withdrawal");

    uint256 aliceClaimableShares2 = vault.claimableRedeemRequest(0, alice);
    uint256 aliceClaimableAssets2 = (aliceClaimableShares2 * price) / 1e36;
    assertEq(aliceClaimableShares2, aliceHalf, "Alice still has claimables (manual path)");
    assertGt(aliceClaimableAssets2, reservedAfter, "manual claimables remain with zero reservation");

    // Optional sanity: vault asset balance decreased by Bob's withdrawal only
    uint256 vaultAssetsAfter = vault.totalAssets();
    assertEq(vaultAssetsBefore - vaultAssetsAfter, bobMax, "vault paid only the reserved portion");
}
```

Recommended Mitigation: Consider making `_fulfillRedeemRequest` the single source of truth for reservation accounting:

1. Move`reservedLiquidity` bump into `_fulfillRedeemRequest`.
2. Remove `reservedLiquidity` increments from `processUpToShares` / `processUpToRequestId` (to avoid double counting).

Accountable: Fixed in commit `c3a7cbf`

Cyfrin: Verified. Recommended mitigation implemented. `reservedLiquidity` is tracked in `_fulfillRedeemRequest` and removed form the "process" functions.

7.3.6 `AccountableFixedTerm::claimInterest` unpredictable due to share burn mechanics

Description: `AccountableFixedTerm::claimInterest` lets a lender redeem their share of already-paid interest by burning vault shares and receiving assets. The burn uses a divisor based on the full-term max net return (fixed at loan acceptance), not the interest actually funded so far:

```
uint256 maxNetYield = PRECISION + _interestParams.netReturn;
claimedInterest = shares.mulDiv(claimableInterest, totalShares, Math.Rounding.Floor);
uint256 usedShares = claimedInterest.mulDiv(PRECISION, maxNetYield, Math.Rounding.Ceil);
```

Because `netReturn` is an optimistic, end-of-term figure, early claimers burn fewer shares per unit claimed, shrinking `totalSupply` and making later outcomes order- and timing-dependent. This yields unpredictable per-user results and creates a systematic advantage for early claimers, especially harmful if the loan later underperforms or defaults, where early claims are crystallized at optimistic rates and late claimers eat the shortfall.

If the loan finishes without default and everyone eventually claims, equal-share lenders converge to the same total interest.

Impact: * Unpredictable user payouts / MEV: Two equal lenders can claim different amounts purely due to claim order; bots can claim immediately after `pay()` to improve their take.

- Asymmetric default risk: If the loan defaults before maturity, early claimers have already extracted cash flows computed using the potential full-term net return. Late/non-claimers are left with less remaining claimable interest/recovery, creating an unfair “claim early” optimization and worsening losses for cooperative users.
- UX / reputational risk: Users pressing “claim” cannot deterministically know the amount; outcomes can be front-run within the same interval.

Proof of Concept: Add the following test to `AccountableFixedTerm.t.sol`:

```
function test_earlyClaimerAdvantage_dueToMaxNetReturnBurn_usdc() public {
    vm.warp(1739893670);

    // Setup borrower/terms identical to other tests
    vm.prank(manager);
    usdcLoan.setPendingBorrower(borrower);

    vm.prank(borrower);
    usdcLoan.acceptBorrowerRole();

    vm.prank(manager);
    usdcLoan.setTerms(
        LoanTerms({
            minDeposit: 0,
            minRedeem: 0,
            maxCapacity: USDC_AMOUNT,
            minCapacity: USDC_AMOUNT / 2,
            interestRate: 1e5,
            interestInterval: 30 days,
            duration: 360 days,
            lateInterestGracePeriod: 2 days,
            depositPeriod: 2 days,
            acceptGracePeriod: 0,
            lateInterestPenalty: 5e2,
            withdrawalPeriod: 0
        })
    );

    // Equal deposits for Alice & Bob
    uint256 userDeposit = USDC_AMOUNT / 2;

    uint256 aliceBalanceBefore = usdc.balanceOf(alice);
    uint256 bobBalanceBefore   = usdc.balanceOf(bob);

    vm.prank(alice);
    usdcVault.deposit(userDeposit, alice, alice);

    vm.prank(bob);
    usdcVault.deposit(userDeposit, bob, bob);

    // Sanity: equal initial shares
    assertEq(usdcVault.balanceOf(alice), userDeposit, "alice initial shares");
    assertEq(usdcVault.balanceOf(bob),   userDeposit, "bob initial shares");

    // Accept loan
    vm.warp(block.timestamp + 3 days);
    vm.prank(borrower);
    usdcLoan.acceptLoanLocked();

    // Fund borrower to pay interest and approve
    usdc.mint(borrower, 2_000_000e6);
    vm.prank(borrower);
    usdc.approve(address(usdcLoan), 2_000_000e6);

    uint256 aliceMidClaim;
    uint256 aliceEndClaim;
    uint256 bobEndClaim;

    // Pay month by month; Alice claims once in the middle, Bob waits
    for (uint8 i = 1; i <= 12; i++) {
        uint256 nextDueDate = usdcLoan.loan().startTime + (i * usdcLoan.loan().interestInterval);
        vm.warp(nextDueDate + 1 days);
```

```
        // Borrower pays owed interest for this interval
        vm.startPrank(borrower);
        uint256 owed = _interestOwed(usdcLoan);
        usdcLoan.pay(owed);
        vm.stopPrank();

        // Alice claims right after month 6 payment
        if (i == 6) {
            vm.prank(alice);
            aliceMidClaim = usdcLoan.claimInterest();
            assertGt(aliceMidClaim, 0, "alice mid-term claim > 0");
        }
    }

    // After last payment, both can claim
    vm.prank(alice);
    aliceEndClaim += usdcLoan.claimInterest();

    vm.prank(bob);
    bobEndClaim += usdcLoan.claimInterest();

    uint256 aliceTotal = aliceMidClaim + aliceEndClaim;
    uint256 bobTotal   = bobEndClaim;

    // Alice has gotten more than Bob by claiming early
    assertGt(aliceTotal, bobTotal, "Alice (mid+end) should claim more than Bob (end only)");

    // repay & clean-up
    vm.prank(borrower);
    usdcLoan.repay(0);

    // Ensure both still redeem principal back pro-rata after interest claims
    uint256 sharesAlice = usdcVault.balanceOf(alice);
    uint256 sharesBob   = usdcVault.balanceOf(bob);

    vm.prank(alice);
    usdcVault.requestRedeem(sharesAlice, alice, alice);
    vm.prank(bob);
    usdcVault.requestRedeem(sharesBob, bob, bob);

    vm.startPrank(alice);
    uint256 maxWithdrawAlice = usdcVault.maxWithdraw(alice);
    usdcVault.withdraw(maxWithdrawAlice, alice, alice);
    vm.stopPrank();

    vm.startPrank(bob);
    uint256 maxWithdrawBob = usdcVault.maxWithdraw(bob);
    usdcVault.withdraw(maxWithdrawBob, bob, bob);
    vm.stopPrank();

    assertEq(usdcVault.balanceOf(alice), 0, "alice no shares");
    assertEq(usdcVault.balanceOf(bob),   0, "bob no shares");

    uint256 aliceBalanceAfter  = usdc.balanceOf(alice);
    uint256 bobBalanceAfter    = usdc.balanceOf(bob);

    uint256 aliceGain = aliceBalanceAfter - aliceBalanceBefore;
    uint256 bobGain   = bobBalanceAfter   - bobBalanceBefore;

    // Alice and Bob has gained the same in the end
    assertEq(aliceGain, bobGain, "alice and bob gained the same");
```

```
}
```

Recommended Mitigation: Consider replacing the share-burn with an accumulator ("rewards-per-share") model: Maintain a high-precision `accInterestPerShare` that increases only when real net interest is paid (after fees) by `netInterest / totalShares`; each lender tracks a checkpoint of this accumulator, and on claim receives `(accCurrent  checkpoint) × shares`, then updates their checkpoint. If transfers/mints/burns were ever allowed mid-loan, first settle pending interest for the party(ies) at the current accumulator and then adjust checkpoints:

```
uint256 accInterestPerShare;
mapping(address user => uint256 index) userIndex;
mapping(address user => uint256 interest) pendingInterest;

function onTransfer(address from, address to, uint256 amount) external onlyVault nonReentrant {

    // Settle sender's pending interest (if not mint)
    if (from != address(0)) {
        _settleAccount(from);
        userIndex[from] = accInterestPerShare;
    }

    // Settle receiver's pending interest (if not burn)
    if (to != address(0)) {
        _settleAccount(to);
        userIndex[to] = accInterestPerShare;
    }

}

/// Internal: settle one account's pending interest using current accumulator
function _settleAccount(address user) internal {
    uint256 shares = vault.balanceOf(user);
    uint256 idx = userIndex[user];

    if (shares == 0) {
        userIndex[user] = accInterestPerShare;
        return;
    }

    uint256 delta  = accInterestPerShare - idx;
    if (delta == 0) return;

    pendingInterest[user] += (shares * delta) / PRECISION;
    userIndex[user] = accInterestPerShare;
}
```

This makes payouts deterministic and call-order independent, distributes only actually received interest (so no "pre-claiming" future yield), and remains fair under partial payments or defaults while preserving price invariance without burning.

Accountable: Fixed in commits 19a50c8 and fd74c1d

Cyfrin: Verified. An interest accrual system is used and the vault now calls an `onTransfer`-hook on the strategy for transfers.

7.3.7 Fees never deducted in `AccountableOpenTerm` loan

Description: In `AccountableOpenTerm`, `interestData()` returns non-zero `performanceFee` and `establishmentFee`, but no path ever charges these fees. `_accrueInterest()` only updates `_scaleFactor` for base interest and none of `supply` or `repay` calls `FeeManager` (unlike FixedTerm's `collect`). As a result, fees are never charged.

Impact: Protocol/manager fees are effectively never taken.

Recommended Mitigation: Consider charge the fee in `supply()/repay()`, before any other state changes. Compute fees for the elapsed period and transfer to `FeeManager`, then proceed.

Accountable: Fixed in commits fce6961 and 8e53eba

Cyfrin: Verified. `performanceFee` and `establishmentFee` are now deducted for open term loans.

7.3.8 Borrower in OpenTerm loan can stay delinquent effectively forever

Description: Delinquency is flagged when vault reserves fall below `_calculateRequiredLiquidity()`, setting `delinquencyStartTime`. Late penalties only accrue after the grace period elapses. If the borrower briefly restores liquidity (e.g., `supply()/repay()`) before grace expiry, delinquency is cleared and `delinquencyStartTime` resets to 0. The borrower can immediately `borrow()` again to drop reserves to the threshold and the next block, when interest has accrued again, start a fresh grace window. This "pulse" can be executed back-to-back, even within one block, allowing the borrower to remain effectively delinquent indefinitely without ever incurring penalties.

Impact: Borrowers can avoid late penalties while keeping lenders under-reserved, degrading lender protections.

Recommended Mitigation: Consider removing the grace period entirely so penalties accrue as soon as the loan becomes delinquent. This would reduce complexity and be in line with how a lot of other lending protocols work.

Alternatively consider redesigning the grace period to be cumulative, i.e. A year loan has a cumulative 1 week grace period which the borrower can draw from.

Accountable: We will acknowledge this. We don't have an actionable path on how loans are managed when it comes to penalties grace periods or even whether penalties are enabled or not. Having a considerable grace period is by design and as a fallback a manager can always initiate a default. In most use-cases borrow/repay actions won't be very often and given these entities deploy funds to other venues, also off-chain, doing such actions can come with a reputational cost.

7.3.9 Withdrawal queue `RequestPrice` can be front run in case of defaults

Description: When `processingMode == ProcessingMode.RequestPrice` in `AccountableWithdrawalQueue`, a redeem request's value is fixed at the request-time share price. The request is later processed potentially at a very different price.

Impact: * Normal operation: Requesters are typically disadvantaged because price usually rises as interest accrues. Locking at request time forfeits subsequent gains.

- Defaults: Requesters can front-run defaults by submitting withdrawals just before delinquency/default and keep the pre-default higher price, draining liquidity and pushing losses onto remaining LPs. This worsens loss socialization precisely when fairness matters most.

Recommended Mitigation: Consider removing `ProcessingMode.RequestPrice` (and `AccountableWithdrawalQueue .processingMode` all together) so redemption value is always determined at processing time. Alternatively implement a safeguard for large price movements that will invalidate the redeem request.

Accontable: Fixed in commit 4e5eef5

Cyfrin: Verified. `processingMode` removed and current price used throughout.

7.3.10 Auto-draw on `AccountableFixedTerm::pay` lets third parties force unwanted borrowing

In `AccountableFixedTerm::pay`, any positive `_loan.drawableFunds` are automatically drawn via `_updateAndRelease(drawableFunds)` before transferring the due interest/fees:

```
uint256 drawableFunds = _loan.drawableFunds;
if (drawableFunds > 0) {
    _updateAndRelease(drawableFunds);
}
```

Since `_loan.drawableFunds` increases when users deposit/mint into the vault, a third party can deposit immediately before the borrower calls `pay`. This causes `pay` to both increase `_loan.outstandingPrincipal` by the new liquidity and also add remaining-term interest on that added principal, while releasing the assets to the borrower, without borrower consent.

Impact: Borrower loses discretion over principal size. Calling `pay` can increase debt (principal + future interest) unexpectedly. This enables griefing/economic DoS as attackers can “stuff” the vault before each payment window, repeatedly forcing draws and increasing interest payments in the future.

Recommended Mitigation: Consider removing auto-draw from `AccountableFixedTerm::pay`. Loan increases should occur only via an explicit borrower action (e.g., `draw(uint256)`), not implicitly during interest payment.

Accountable: Fixed in commit 03f871b

Cyfrin: Verified. "auto-draw" is removed from `pay`.

7.3.11 Frequent `AccountableOpenTerm::accrueInterest` calls reduce interest accrual

Description: In `AccountableOpenTerm::_linearInterest`, interest accrual uses integer math, `_linearInterest(rate, dt) = rate * dt / DAYS_360_SECONDS`:

```
function _linearInterest(uint256 interestRate, uint256 timeDelta) internal pure returns (uint256) {
    return interestRate.mulDiv(timeDelta, DAYS_360_SECONDS);
}
```

For small `timeDelta`, this often rounds to zero. Yet `accrueInterest()` still sets `_accruedAt = block.timestamp` even when the computed increment is zero. Repeated calls with short intervals therefore discard elapsed time in many zero-increment slices, producing a persistently lower `_scaleFactor` than a single accrual over the same wall-clock period.

For example, for a 15% APY (150_000) you would have to call once every 207 seconds (~4 minutes):

```
360 days / 150_000 = 31104000 / 150_000 = 207
```

Impact: Any actor can repeatedly call `accrueInterest()` at short intervals to suppress interest growth. Over time this materially underpays LPs (lower share price / fewer assets owed by the borrower) and reduces protocol fee bases tied to interest. The effect compounds with call cadence and APR, creating measurable loss without needing privileged access.

Proof of Concept: Add the following test to `AccountableOpenTerm.t.sol`:

```
function test_interest_rounding_from_frequent_accrue_calls() public {
    vm.warp(1739893670);

    vm.prank(manager);
    usdcLoan.setPendingBorrower(borrower);
    vm.prank(borrower);
    usdcLoan.acceptBorrowerRole();

    // Use a common APR (15%) and short interval; depositPeriod = 0 to keep price logic simple.
    LoanTerms memory terms = LoanTerms({
        minDeposit: 0,
        minRedeem: 0,
        maxCapacity: USDC_AMOUNT,
        minCapacity: USDC_AMOUNT / 2,
        interestRate: 150_000,        // 15% APR in bps units
        interestInterval: 30 days,
        duration: 0,
        depositPeriod: 0,
        acceptGracePeriod: 0,
        lateInterestGracePeriod: 0,
        lateInterestPenalty: 0,
        withdrawalPeriod: 0
```

```
    });
    vm.prank(manager);
    usdcLoan.setTerms(terms);
    vm.prank(borrower);
    usdcLoan.acceptTerms();

    // Provide principal so interest accrues on outstanding assets.
    vm.prank(alice);
    usdcVault.deposit(USDC_AMOUNT, alice, alice);

    // Snapshot the baseline state just after start.
    uint256 snap = vm.snapshot();

    // -----------------------------------------------------------
    // Scenario A: "Spam accrual" - call accrueInterest() every 12s for 1 hour.
    // Each 12s step yields baseRate = rate * 12 / 360d  0 (integer), but _accruedAt is reset,
    // so we lose that fractional time forever.
    // -----------------------------------------------------------
    uint256 step = 180;          // 3 minutes
    uint256 total = 3600;        // 1 hour
    uint256 n = total / step;    // 300 iterations

    for (uint256 i = 0; i < n; i++) {
        vm.warp(block.timestamp + step);
        usdcLoan.accrueInterest(); // returns new scale but we just trigger the reset
    }

    // Capture the resulting scale factor after the spammy accrual pattern
    uint256 sfSpam = usdcLoan.accrueInterest(); // one more call just to read the value

    // -----------------------------------------------------------
    // Scenario B: Single accrual after the same total wall-clock time.
    // -----------------------------------------------------------
    vm.revertTo(snap);
    vm.warp(block.timestamp + total);
    uint256 sfClean = usdcLoan.accrueInterest();

    // Expect the spammed path to have strictly lower scale factor than the clean path.
    assertLt(sfSpam, sfClean, "frequent zero-delta accrual bleeds interest vs single accrual");

    // Anything more often than 207 in this case will result in no interest growth at all.
    assertEq(sfSpam, 1e36, "frequent accruals yield no interest growth");
}
```

Recommended Mitigation: Consider using higher precision to track interest rate. For example, 1e18 or 1e36.

Accountable: Fixed in commit 29c3f72

Cyfrin: Verified. `_linearInterest` now scales with `PRECISION`.

7.3.12 Invalid `maxWithdraw()` check in `withdraw()`

Description: Vault incorrectly checks `maxWithdraw(receiver)` instead of `maxWithdraw(controller/owner)`.

Impact:

- Allows unauthorized withdrawals by exploiting the receiver's limits instead of the owner's.
- DDoS in `withdraw()`

Proof of Concept: Violated: https://prover.certora.com/output/52567/ef88bd2d76b74cafb175f8d026e484b3/?anonymousKey=599db11fbc5df1632ff4006c69a03f836b23fa6c

```
// MUST NOT be higher than the actual maximum that would be accepted
```

```
rule eip4626_maxWithdrawNoHigherThanActual(env e, uint256 assets, address receiver, address owner) {

    setup(e);

    storage init = lastStorage;

    mathint limit = maxWithdraw(e, owner) at init;

    withdraw@withrevert(e, assets, receiver, owner) at init;
    bool reverted = lastReverted;

    // Withdrawals above the limit must revert
    assert(assets > limit => reverted, "Withdraw above limit MUST revert");
}
```

Verified after the fix: https://prover.certora.com/output/52567/8e7cfdf612d64a4cb7e5d9d9d939968e/?anonymousKey=a961467ded443bd1cab3718ca882be71f38887e9

Recommended Mitigation:

```
diff --git a/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
↪ b/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
index a64f47c..c8824bb 100644
--- a/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
+++ b/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
@@ -173,7 +173,7 @@ contract AccountableAsyncRedeemVault is IAccountableAsyncRedeemVault, Accountabl
     function withdraw(uint256 assets, address receiver, address controller) public onlyAuth returns
     ↪   (uint256 shares) {
         _checkController(controller);
         if (assets == 0) revert ZeroAmount();
-        if (assets > maxWithdraw(receiver)) revert ExceedsMaxRedeem();
+        if (assets > maxWithdraw(controller)) revert ExceedsMaxRedeem(); // @certora FIX for
↪   eip4626_maxWithdrawNoHigherThanActual (receiver -> controller)

         VaultState storage state = _vaultStates[controller];
         shares = _convertToShares(assets, state.withdrawPrice, Math.Rounding.Floor);
```

Accountable: Fixed in commit 6dc92b0

Cyrfin: Verified. `controller` now passed to `maxWithdraw`.

7.4 Low Risk

7.4.1 Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collisions

Description: The protocol has upgradeable contracts which other contracts inherit from. These contracts should either use:

- ERC7201 namespaced storage layouts - example
- storage gaps (though this is an older and no longer preferred method)

The ideal mitigation is that all upgradeable contracts use ERC7201 namespaced storage layouts.

Without using one of the above two techniques storage collision can occur during upgrades.

Accountable: Fixed in commit 8422762

Cyfrin: Verified. Namespaced storage now used in `AccountableStrategy`.

7.4.2 Missing controller validation in `AccountableAsyncRedeemVault::requestRedeem` allows zero address state

Description: The `requestRedeem()` function fails to call `_checkController(controller)` validation, allowing the zero address to accumulate vault state.

Impact: `zeroControllerEmptyState` violation.

Proof of Concept: Violated: https://prover.certora.com/output/52567/acc42433123e4b289c0f84e69fa52a44/?anonymousKey=e60b3d66b5574868073bfde4218b385aa2fe5f2a

```
// Zero address must have empty state for all vault fields
invariant zeroControllerEmptyState(env e)
    ghostVaultStatesMaxMint256[0] == 0 &&
    ghostVaultStatesMaxWithdraw256[0] == 0 &&
    ghostVaultStatesDepositAssets256[0] == 0 &&
    ghostVaultStatesRedeemShares256[0] == 0 &&
    ghostVaultStatesDepositPrice256[0] == 0 &&
    ghostVaultStatesMintPrice256[0] == 0 &&
    ghostVaultStatesRedeemPrice256[0] == 0 &&
    ghostVaultStatesWithdrawPrice256[0] == 0 &&
    ghostVaultStatesPendingDepositRequest256[0] == 0 &&
    ghostVaultStatesPendingRedeemRequest256[0] == 0 &&
    ghostVaultStatesClaimableCancelDepositRequest256[0] == 0 &&
    ghostVaultStatesClaimableCancelRedeemRequest256[0] == 0 &&
    !ghostVaultStatesPendingCancelDepositRequest[0] &&
    !ghostVaultStatesPendingCancelRedeemRequest[0] &&
    ghostRequestIds128[0] == 0
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

Verified after the fix: https://prover.certora.com/output/52567/f385fd34e82c4635bd410279e4da2c97/?anonymousKey=82309551a07845692bfabb2164179224523f87ba

Recommended Mitigation:

```
diff --git a/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
↪  b/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
index 4cd0a3e..a64f47c 100644
--- a/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
+++ b/credit-vaults-internal/src/vault/AccountableAsyncRedeemVault.sol
@@ -113,6 +113,9 @@ contract AccountableAsyncRedeemVault is IAccountableAsyncRedeemVault, Accountabl
         onlyAuth
         returns (uint256 requestId)
     {
+        // @certora FIX for zeroControllerEmptyState
```

```
+        _checkController(controller);
+
         _checkOperator(owner);
         _checkShares(owner, shares);
```

Accountable: Fixed in commit e90d3de

Cyfrin: Verified. `checkController` added as a modifier to the function.

7.4.3 Reserved assets could be extracted from the Vault

Description: Some strategy functions can release assets without checking if those assets are part of `reservedLiquidity`. `AccountableFixedTerm._loan.drawableFunds` is not verified to be in sync with the queue `reservedLiquidity`. Hence the borrower can inadvertently borrow more funds than they should.

Impact: The vault can become insolvent by releasing funds needed to honor a withdrawal.

Proof of Concept: Violated in `FixedTerm.acceptLoanLocked()`, `FixedTerm.borrow()`, `FixedTerm.pay()`, `FixedTerm.acceptLoanDynamic()`, `FixedTerm.claimInterest()`: https://prover.certora.com/output/52567/edb399a43d1849a9b22f027e66b17924/?anonymousKey=3dcf62dfa004381083966b3639b6a485fa2e9501

```
// Reserved liquidity must not exceed total assets
invariant reservedLiquidityBacked(env e)
    ghostReservedLiquidity256 <= ghostTotalAssets256
```

Recommended Mitigation: When `reservedLiquidity` is increased in the withdrawal queue, this needs to be synced to the FixedTerm starategy.

Accountable: Fixed in commit 979c0e.

Issue was addressed to satisfy the invariant and prevent future upgrades that might allow redemptions in a FixedTerm loan, but as of right now there's no possible way to increase `reservedLiquidity` such that it is out-of-sync with`drawableFunds`.

Borrowing after the loan is in a `Repaid` state cannot happen due to `_requireLoanOngoing` so any redemptions that increase `reservedLiquidity` would require a state when both depositing/borrowing is blocked.

Cyfrin: Verified. `reservedLiquidity` is now checked in FixedTerm.

7.4.4 `Authorizable::_verify` should use EIP-712 typed structured data hashing

Description: `Authorizable::_verify` signs ad-hoc payloads that include `chainId`, but the flow is not EIP-712 typed-data compliant. This limits wallet UX/visibility and interoperability and mixes domain data (`chainId`) with message data.

Impact: Users are more susceptible to ambiguous signing prompts; weaker ecosystem compatibility; harder audits/upgrades; higher risk of encoding/packing mistakes and replay bugs across contracts or chains.

Recommended mitigation: Adopt EIP-712 and move `chainId` to the domain separator (remove it from the struct). Keep the existing intent of the message:

- **Domain:** `{ name: "Authorizable", version: "1", chainId, verifyingContract: address(this) }`.
- **Typed struct (no chainId inside):**

```
struct TxAuthData {
    bytes   functionCallData;    // selector + encoded args
    address contractAddress;    // target contract (can be redundant with domain; decide and
    ↪   document)
    address account;            // controller / signer subject
    uint256 nonce;              // per-account nonce
    uint256 blockExpiration;    // deadline
}
```

```
bytes32 constant TXAUTH_TYPEHASH = keccak256(
    "TxAuthData(bytes functionCallData,address contractAddress,address account,uint256
    ↪  nonce,uint256 blockExpiration)"
);
```

- **Hashing & verify (using OZ EIP712 + SignatureChecker):**

```
bytes32 structHash = keccak256(abi.encode(
    TXAUTH_TYPEHASH,
    keccak256(txAuth.functionCallData), // hash dynamic bytes
    txAuth.contractAddress,
    txAuth.account,
    txAuth.nonce,
    txAuth.blockExpiration
));
bytes32 digest = _hashTypedDataV4(structHash);
require(
    SignatureChecker.isValidSignatureNow(signer, digest, signature),
    "INVALID_SIGNATURE"
);
```

Accountable: Fixed in commit 70cd486

Cyfrin: Verified. EIP-712 typed data is now used for the signatures.

7.4.5 Deployment script requires unencrypted private key

Description: The deployment scripts FactoryScript.s.sol and FeeManagerScript.s.sol requires a private key to be stored in clear text as an environment variable:

```
uint256 deployerPk = vm.envUint("DEPLOYER_TESTNET_PK");
```

Storing private keys in plain text represents an operational security risk, as it increases the chance of accidental exposure through version control, misconfigured backups, or compromised developer machines.

A more secure approach is to use Foundry's wallet management features, which allow encrypted key storage. For example, a private key can be imported into a local keystore using `cast`:

```
cast wallet import deployerKey --interactive
```

This key can then be referenced securely during deployment:

```
forge script script/Deploy.s.sol:DeployScript \
    --rpc-url "$RPC_URL" \
    --broadcast \
    --account deployerKey \
    --sender <address associated with deployerKey> \
    -vvv
```

And used just with `vm.startBroadcast()`:

```
vm.startBroadcast();

...

vm.stopBroadcast();
```

For additional guidance, see this explanation video by Patrick.

Accountable: Fixed in commit 79d8cfd

Cyfrin: Verified. Deploy scripts now don't require a private key in clear text.

7.5 Informational

7.5.1 Prevent accidental ownership and admin renouncement

Description: The inherited `renounceOwnership()` and allow the last authority to remove themselves, potentially leaving the contract permanently ownerless or admin-less, blocking critical functions.

Consider override `renounceOwnership()` in `TokenAirdrop` to always revert.

Accountable: Fixed in commit `be75091`

Cyfrin: Verified.

7.5.2 Consider consistently use `Ownable2Step`

Description: Currently some contracts use just Ownable, consider have all contracts use Ownable2Step to prevent accidental ownership loss.

Accontable: Fixed in commit `be75091`

Cyfrin: Verified.

7.5.3 Consider enforcing a minimum deposit amount

The vault/strategy accepts arbitrarily small deposits (down to 1 wei). While functionally correct, dust deposits are effectively useless for legitimate users (since the gas to call `deposit` often exceeds the value deposited) and can be abused by adversaries to abuse rounding edge cases.

To remove a possible attack vector for black hats, consider enforcing a `minimumDepositAmount`.

Accountable: Fixed in `b9edb2b`

Cyfrin: Verified.

7.5.4 Violations of ERC7540 specs

Description: Several deviations from the ERC7540 specs have been noticed for `AccountableAsyncRedeemVault`

1. According to ERC-7540 specification:

   Redeem Request approval of shares for a msg.sender NOT equal to owner may come either from ERC-20 approval over the shares of owner or if the owner has approved the msg.sender as an operator.

The current implementation in `requestRedeem()` only supports operator approval (from owner to caller). The contract does not implement the ERC-20 allowance path for share approval, limiting the request redeem functionality to only operator-approved addresses.

2. As per the EIP,

   All requests with the same requestID MUST transition from Pending to Claimable state at the same time, and receive the same exchange rate

This means the request should always gets processed in full. Right now, the vault implementation allows partial redemptions and that too at different share prices (if processingMode == CurrentPrice).

Impact: Non-compliance with ERC7540.

Recommended Mitigation: Consider documenting if the vault is intended to be completely compliant with the EIP, and if so, consider changing the implementation accordingly.

Accountable: We acknowledge this as it's not our intention to be 100% compliant, we will document this.

7.5.5 Incorrect event emission is possible in `AccountableAsyncRedeemVault::cancelRedeemRequest` flows

Description: `cancelRedeemRequest()` takes "requestID" as input, but it is never used and never validated to be associated with the input controller address.

All cancellation flows (immediate/ async) work with the requestID of the controller address stored in `_requestIds[controller]`, but the input requestID is only used for event data in `CancelRedeemRequest()` and `CancelRedeemClaimable()` events.

Because this is never verified, caller can input any requestID and have it emitted in the events.

Impact: Incorrect event emission is possible, potentially leading to data corruption for the frontend and anyone else using this event data.

Recommended Mitigation: Remove the "requestID" parameter from the `cancelRedeemRequest()` function definition and simply use the existing requestID of the controller in event emission.

Accountable: Fixed in commits `aa64491` and `0675c3d`.

Cyfrin: Verified. The redeem request of the controller is now used.

7.5.6 ERC20 zero amount transfer rejection

Description: The `_checkTransfer` function reverts on zero-amount transfers, violating ERC-20 standard which mandates that transfers of 0 values MUST be treated as normal transfers.

Impact: Violation of `eip20_transferSupportZeroAmount` and `eip20_transferFromSupportZeroAmount`.

Proof of Concept: Violated: https://prover.certora.com/output/52567/9c9c3c73f4d64f9baf1284ced4f4a8f5/?anonymousKey=160f0b0d10e3f688f1981708e4aa3819e7023a80

```
// EIP20-06: Verify transfer() handles zero amount transfers correctly
// EIP-20: "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event."
rule eip20_transferSupportZeroAmount(env e, address to, uint256 amount) {

    setup(e);

    // Perform transfer
    transfer(e, to, amount);

    // Zero amount transfers must succeed
    satisfy(amount == 0);
}

// EIP20-09: Verify transferFrom() handles zero amount transfers correctly
// EIP-20: "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event."
rule eip20_transferFromSupportZeroAmount(env e, address from, address to, uint256 amount) {

    setup(e);

    // Perform the transferFrom
    transferFrom(e, from, to, amount);

    // Zero amount transferFrom must succeed
    satisfy(amount == 0);
}
```

Verified after the fix: https://prover.certora.com/output/52567/0babf2c2b4da49ec87cc0ae00036b0e7/?anonymousKey=21b1c4b60901ee2fea0115aa8a1b0e621c04bfaa

Recommended Mitigation:

```
diff --git a/credit-vaults-internal/src/vault/AccountableVault.sol
↪  b/credit-vaults-internal/src/vault/AccountableVault.sol
index 629b6d0..fb3676a 100644
```

```diff
--- a/credit-vaults-internal/src/vault/AccountableVault.sol
+++ b/credit-vaults-internal/src/vault/AccountableVault.sol
@@ -141,7 +141,8 @@ abstract contract AccountableVault is IAccountableVault, ERC20, AccessBase {

     /// @dev Checks transfer restrictions before executing the underlying transfer
     function _checkTransfer(uint256 amount, address from, address to) private {
-        if (amount == 0) revert ZeroAmount();
+        // @certora FIX for eip20_transferSupportZeroAmount and eip20_transferFromSupportZeroAmount
+        // if (amount == 0) revert ZeroAmount();
         if (!transferableShares) revert SharesNotTransferable();
         if (!isVerified(to, msg.data)) revert Unauthorized();
         if (throttledTransfers[from] > block.timestamp) revert TransferCooldown();
```

Accountable: Fixed in commit e90d3de

Cyfrin: Verified.

7.5.7 `nonReentrant` is not the first modifier

Description: In `FeeManager::withdrawProtocolFee`, `nonReentrant` is not the first modifier. To protect against reentrancy in other modifiers, the `nonReentrant` modifier should be the first modifier in the list of modifiers. Consider putting `nonReentrant` first for consistent reentrancy protection.

Accountable: Fixed in commit c7f31b5

Cyfrin: Verified.

7.5.8 Unused errors

The following errors in) `https://github.com/Accountable-Protocol/audit-2025-09-accountable/blob/fc43546fe67183235c0725f6214ee2b876b1aac6/src/constants/Errors.sol` are unused. Consider using or removing the unused error.

- Line: 15 `error InvalidVerifier();`
- Line: 18 `error InvalidExpiration();`
- Line: 27 `error UnauthorizedOwnerOrReceiver();`
- Line: 30 `error UnauthorizedController();`
- Line: 45 `error AccountAlreadyVerified();`
- Line: 49 `error NotAdminOrOperator(address account);`
- Line: 52 `error Paused(address account);`
- Line: 59 `error CancelDepositRequestPending();`
- Line: 65 `error DepositRequestWasCancelled();`
- Line: 71 `error ExceedsDepositLimit();`
- Line: 89 `error NoDepositRequest();`
- Line: 95 `error NoPendingDepositRequest();`
- Line: 101 `error NoCancelDepositRequest();`
- Line: 113 `error ProposalExpired();`
- Line: 116 `error NoPendingProposal();`
- Line: 122 `error AlreadyInQueue();`
- Line: 141 `error LoanAlreadyAccepted();`
- Line: 153 `error LoanInDefault();`

- Line: 162 `error LoanNotAcceptedByBorrower();`
- Line: 168 `error ZeroSharePrice();`
- Line: 177 `error LoanNotRepaid();`
- Line: 186 `error PaymentNotDue();`
- Line: 192 `error RequestDepositFailed();`
- Line: 195 `error RequestRedeemFailed();`
- Line: 210 `error BorrowerNotSet();`
- Line: 213 `error PriceOracleNotSet();`
- Line: 216 `error RewardsDistributorNotSet();`

Accountable: Errors removed in commit 18ce919

Cyfrin: Verified.

7.5.9 State changes without events

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

- Line: 47

```
function setSecurityAdmin(address securityAdmin_) external onlyOwner {
```

- Line: 53

```
function setOperationsAdmin(address operationsAdmin_) external onlyOwner {
```

- Line: 59

```
function setTreasury(address treasury_) external onlyOwner {
```

- Line: 65

```
function setVaultFactory(address vaultFactory_) external onlyOwner {
```

- Line: 71

```
function setRewardsFactory(address rewardsFactory_) external onlyOwner {
```

Accountable: Fixed in commit 13600f4

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 Storage read optimizations

Description:

1. `AccountableOpenTerm::_calculateRequiredLiquidity`: `vault` and `_scaleFactor` can be cached. Also consider changing so that `_calculateRequiredLiquidity` takes `address vault_` as a parameter. That would allow to cache the `vault` read in the `_isDelinquent`, `_getAvailableLiquidity`, `_borrowable` and `_validateLiquidityForTermChange` flows as well.

2. `AccountableOpenTerm::_getAvailableLiquidityForProcessing`: `vault` can be cached. Also consider same as above, add `address vault_` as a parameter, then use a cached value from `_processAvailableWithdrawals`.

3. `AccountableOpenTerm::_penaltyFee`, use the cached value `gracePeriod` on L595

4. `AccountableOpenTerm::supply`: `vault` can be cached

5. `AccountableOpenTerm::repay`: `vault` can be cached.

6. `AccountableFixedTerm::_sharePrice`: `loanState` can be cached.

7. `AccountableStrategy::acceptBorrowerRole`: Use `msg.sender` instead of `pendingBorrower` on L185 and instead of `borrower` on L188

8. `AccountableStrategy::_requireLoanNotOngoing`: `loanState` can be cached.

9. `AccountableStrategy::_requireLoanOngoing`: `loanState` can be cached.

10. AccountableWithdrawalQueue::_push: Set `_queue.nextRequestId` to `1` at construction and remove the if to save a read each `_push`.

11. `AccountableAsyncRedeemVault::redeem`: `state.redeemPrice` can be cached

12. `AccountableAsyncRedeemVault::withdraw`: `state.withdrawPrice` can be cached.

13. `AccountableAsyncRedeemVault::_updateRedeemState`: `state.maxWithdraw` and `state.redeemShares` can be cached.

14. `AccountableAsyncRedeemVault::_fulfillRedeemRequest`: `state.pendingRedeemRequest`, `state.maxWithdraw` and `state.redeemShares` can be cached.

15. `AccountableAsyncRedeemVault::maxRedeem` and AccountableAsyncRedeemVault::maxWithdraw: Can be rewritten as:

```
function maxWithdraw(address receiver) public view override returns (uint256 maxAssets) {
    VaultState storage state = _vaultStates[receiver];
    maxAssets = state.maxWithdraw;
    if (state.redeemShares == 0) return 0;
}
```

16. `Authorizable::_verify`: `signer` can be cached.

17. `RewardsDistributorMerkle::acceptRoot`: `_pendingRoot.validAt` can be cached.

18. `RewardsDistributorMerkle::claim`: `claimed[account][asset])` can be cached

19. `RewardsDistributorStrategy::claim`: `claimed[account][asset])` can be cached

Accountable: Most fixed in commit 8e1cfa2

Cyfrin: Verified.