Accountable PR50 Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

MrPotatoMagic

January 30, 2026

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 PR-50 extends the existing FixedTerm/OpenTerm lending system with a DVNPublisher-based oracle pipeline and a new AccountableYield strategy that depends on external NAV updates. A set of authorized signers submit signed `PublishRequest`s (value + timestamp + batch id + nonce), which accumulate in the DVNPublisher's pending set. An authorized executor then calls `execute()` to filter stale/invalid requests, de-duplicate by signer (keeping the signer's latest value), compute a median, enforce an optional max deviation vs. the previous batch, and finally call `strategy.publishRate(medianValue)` to update the strategy's reference value.

On the strategy side, AccountableOpenTerm continues to use `_scaleFactor` to linearly accrue interest and exposes `publishRate(newRate)` gated to the configured `dvnPublisher`, making interest-rate updates asynchronous (propose/collect, execute, publish). AccountableYield is introduced as a NAV-based yield strategy where `publishRate()` is used to publish the latest deployed/NAV value from the DVN publisher/oracle; it adds a configurable NAV grace period (default 24h, min 1h) and uses a "not stale" gating model for selected actions.

4.1 Actors and Roles

- **DVN Signers:** Approved addresses that sign `PublishRequest`s (value + timestamp + batch id + nonce) for inclusion in DVNPublisher.
- **DVN Executor:** Approved address that runs `execute()` to compute the median and push it on-chain to the strategy.
- **Manager:** Configures DVNPublisher (signers/executors/threshold/maxStaleness/maxDeviation) and can `cancelBatch()` to clear pending requests.
- **Strategy (OpenTerm / Yield):** Accepts DVNPublisher-delivered updates via `publishRate(...)` (restricted to `dvnPublisher`).

4.2 Key Components

- **DVNPublisher:**

- **publish(request, signature):** verifies batch id, nonce, signer approval, and rejects future timestamps; appends to `_pendingRequests` up to a fixed cap.
- **execute():** filters out stale requests (`timestamp + maxStaleness`), ignores removed signers, de-duplicates signers (latest value wins), computes median, optionally enforces maxDeviation vs prior published median, stores batch data, clears pending requests, increments batch id, then calls `strategy.publishRate(medianValue)`.
- **cancelBatch():** manager clears `_pendingRequests` to recover from malformed/stale batches without advancing the batch id.

- **AccountableYield:**
  - Introduces a NAV-driven strategy with a NAV grace period (`DEFAULT_NAV_GRACE_PERIOD = 24 hours`, `MIN_NAV_GRACE_PERIOD = 1 hour`) and a `whenNotStale` modifier pattern to block certain flows when updates are stale.
  - Requires a configured `dvnPublisher` to be set before terms can be set (i.e., DVN/NAV oracle is a first-class dependency for operating the strategy).

4.3 Centralization and Oracle Risk

PR-50 increases reliance on off-chain measurement + multi-signer attestation + privileged execution: signers determine the submitted values, executors determine when a batch is finalized, and the manager can revoke signers/executors or cancel batches. While median + staleness/deviation checks help bound some failure modes, operational security around signer keys, executor availability, and governance of DVNPublisher parameters becomes critical because `execute()` directly drives strategy state via `publishRate()`.

5 Audit Scope

The audit scope was limited to PR50:

```
// new files
src/batcher/AtomicBatcher.sol
src/factory/DVNPublisherFactory.sol
src/factory/YieldStrategyFactory.sol
src/publisher/DVNPublisher.sol
src/strategies/AccountableYield.sol
src/strategies/storage/YieldStorage.sol

// make upgradeable, fee-related changes
src/modules/FeeManager.sol

// fee-related changes, new function `_managementFeeFactor`
src/strategies/AccountableFixedTerm.sol

// rollback window, new functions related to DVNPublisher
// fee-related changes, new function `onCancelRedeemRequest`
src/strategies/AccountableOpenTerm.sol

// very minor changes, modifier `onlyOperationsAdmin`
src/strategies/AccountableStrategy.sol

// very minor changes - new storage slot
src/strategies/storage/OpenTermStorage.sol
src/strategies/storage/StrategyStorage.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Accountable PR50 smart contracts provided by Accountable. In this period, a total of 22 issues were found.

During this PR50 follow-up audit, we identified five medium-severity issues spanning the newly introduced DVN-driven NAV/rate update flows and queue/withdrawal batching. These included (i) NAV staleness protections being bypassable via public queue-processing entrypoints, (ii) repay vs. DVN NAV publish ordering that could temporarily "undo" repayment accounting, and (iii) an OpenTerm batch-cancellation accounting bug that could delay earlier lenders' withdrawals; plus factory hardening gaps that could allow deployment of unverified Yield strategies.

We also reported six low-severity findings mainly around state/accounting correctness and liveness edge cases (e.g., delinquency refresh timing in hooks and DVN flows, batch-mode transitions when `withdrawalPeriod` changes, and duplicate-signer handling nuances in DVN execution). Finally, we included four informational notes (validation/UX hardening around rate proposals, NAV grace-period errors, and DVN update age checks) and seven gas optimizations focused on precomputations, avoiding redundant loads, and minor event/emission cleanups.

Summary

Project Name	Accountable PR50
Repository	credit-vaults-internal
Commit	ba1c7754f891...
Fix Commit	835f597bd291...
Audit Timeline	Jan 26th - Jan 30th, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	5
Low Risk	6
Informational	4
Gas Optimizations	7
Total Issues	22

Summary of Findings

[M-1] `AtomicBatcher` uses placeholder ERC-7201 namespace	Resolved
[M-2] Immediate withdrawals possible even when NAV is stale through `AccountableYield::accrueAndProcess`	Resolved

[M-3] `AccountableYield::repay` vs `publishRate` transaction ordering can undo repayment accounting	Resolved
[M-4] Cancelling a later-batch request in `AccountableOpenTerm` can delay earlier withdrawals	Resolved
[M-5] Missing modifiers on `YieldStrategyFactory.createYieldStrategy` can lead to deployment of unverified strategies	Resolved
[L-1] `AccountableOpenTerm` rate publish/rollback does not refresh delinquency status	Resolved
[L-2] Fee structure updates can trigger accrual after loan has ended	Resolved
[L-3] Delinquency status update in `AccountableOpenTerm` hooks uses pre-queue state	Resolved
[L-4] Increasing `AccountableOpenTerm.loan.withdrawalPeriod` from `0` can cause withdrawals to become stuck	Resolved
[L-5] Function `execute` overwrites seenSigner values irrespective of request age	Resolved
[L-6] `lastTotalAssets` stores stale value due to update before penalty accrual	Resolved
[I-1] `AccountableOpenTerm` manual interest rate proposal is unbounded	Resolved
[I-2] `AccountableYield::setNavGracePeriod` uses `Unauthorized` error for invalid input	Resolved
[I-3] `DVNPublisher::publish` does not enforce a maximum age for updates	Resolved
[I-4] Consider reverting in `publishedDataByBatchId` for invalid batch IDs	Resolved
[G-1] Precompute `baseSlot` in `AtomicBatcher::_getNonceSlot`	Resolved
[G-2] Precompute `callTypeHash` in `AtomicBatcher::_hashCallArray`	Resolved
[G-3] Reuse `fm` Instead of re-instantiating `IFeeManager` in `AccountableOpenTerm::_mintFeeShares`	Resolved
[G-4] Reuse `aum_` in `_accrueFeeShares` to avoid recomputing debt	Resolved
[G-5] Only update `deployedAssets` when `remaining > 0` in `AccountableYield::repay`	Resolved
[G-6] Consider removing redundant zero address check from `createYieldStrategy`	Resolved
[G-7] Consider emitting events early to save gas	Resolved

7 Findings

7.1 Medium Risk

7.1.1 `AtomicBatcher` uses placeholder ERC-7201 namespace

Description: `AtomicBatcher` derives its nonce storage slot from an ERC-7201 namespace constant that is still a placeholder:

```
/// @notice ERC-7201 namespace for nonce storage
string private constant _NAMESPACE = "<namespace>";
```

If this is not replaced with a unique, project-specific namespace, different contracts/tools that reuse the same placeholder can end up writing to the same storage slot.

Impact: Nonce storage can collide with other code using the same placeholder namespace, potentially breaking replay protection (unexpected nonce changes), causing failed executions, or enabling cross-application interference when running in shared storage contexts (e.g., EIP-7702 style execution in an EOA's storage).

Recommended Mitigation: Replace `"<namespace>"` with a unique, stable identifier (e.g., `"accountable.atomicbatcher.nonce.v1"`), and treat it as immutable across upgrades/deployments.

Accountable: Fixed in commit 2247cec

Cyfrin: Verified. Namespace is now `accountable.atomicbatcher.nonce.v1`.

7.1.2 Immediate withdrawals possible even when NAV is stale through `AccountableYield::accrueAndProcess`

Description: When NAV is stale, `AccountableYield::onRequestRedeem` disables "instant fulfill" by forcing requests into the queue:

```
canFulfill = liquidity >= assets && !_navIsStale();
```

However, `AccountableYield::accrueAndProcess` is publicly callable and is not gated by `whenNotStale`. It processes the withdrawal queue immediately:

```
function accrueAndProcess() external ... {
    _accrueFees();
    usedAssets = _processAvailableWithdrawals();
    _updateDelinquentStatus();
}
```

As a result, a user can queue a redeem request and then immediately call `accrueAndProcess()` (potentially in the same transaction via a router/multicall) to have the request processed even while NAV is stale.

Impact: This undermines the intended protection of "no immediate withdrawals when NAV is stale." Withdrawals can still be processed at the last known (stale) NAV-derived price, which may be economically incorrect during periods when NAV updates are unavailable.

Recommended Mitigation: Gate queue processing while NAV is stale (e.g., add `whenNotStale` to `accrueAndProcess()` and any other public entrypoints that trigger `_processAvailableWithdrawals()`, like `AccountableYield::repay`)

Accountable: Fixed in commit ddcbfa5

Cyfrin: Verified. Both `repay` and `accrueAndProcess` now have the `whenNotStale` modifier.

7.1.3 `AccountableYield::repay` vs `publishRate` transaction ordering can undo repayment accounting

Description: `AccountableYield::repay` reduces `deployedAssets` based on the repaid amount:

```
uint256 deployed = deployedAssets;
deployedAssets -= Math.min(remaining, deployed);
```

But `publishRate(uint256 newDeployedValue)` later overwrites `deployedAssets` with the DVN-reported value:

```
uint256 oldValue = deployedAssets;
deployedAssets = newDeployedValue;
```

Because these are independent transactions, ordering matters. If `repay()` executes first (reducing `deployedAssets`), and then `publishRate()` executes with a value that still reflects the pre-repayment NAV, the overwrite can effectively "reverse" the accounting effect of the borrower's repayment by setting `deployedAssets` back up.

Impact: Transaction ordering can materially change outcomes. In congested conditions, a DVN update can effectively "undo" the accounting effect of a borrower repayment by resetting `deployedAssets` upward, impacting reported NAV/share price, fee accrual, and whether the loan can reach a fully repaid state. This risk is amplified when the NAV grace period is configured to be short (default is 24h, but it can be as low as 1h), increasing the frequency/urgency of updates and making collisions more likely. It is further increased by the DVNPublisher's async publish/execute flow, since there is an inherent delay between when a value is proposed and when it is executed on-chain, making it more likely that repayments occur in between.

Recommended Mitigation: `DVNPublisher.PublishRequest` already includes a `timestamp`, pass that through to `AccountableYield.publishRate` (e.g., `publishRate(uint256 value, uint256 measuredAt)`) and store `lastNavMeasuredAt`. Reject/ignore updates with `measuredAt <= lastNavMeasuredAt` and/or `measuredAt < lastRepayTime` so a stale snapshot cannot overwrite newer repayment accounting.

Accountable: Fixed in commits 5b6498a, 6756c97 and 06b6c4c

Cyfrin: Verified. Code now compares to the timestamp of the measurement, if the value is a mean of the two middle measurements, the older timestamp is used.

7.1.4 Cancelling a later-batch request in `AccountableOpenTerm` can delay earlier withdrawals

Description: The Vault allows cancelling any queued redeem request via `AccountableAsyncRedeemVault::cancelRedeemRequest(controller, receiver)`, which removes that controller's queued shares.

To keep batching metadata in sync, the Vault calls the strategy hook `AccountableOpenTerm::onCancelRedeemRequest(...)`. However, the strategy reduces batch totals starting from `pendingBatch` (oldest batch) and walks forward, without knowing which batch the cancelled request actually belonged to:

```
uint256 batch = pendingBatch;
...
while (shares > 0 && batch <= maxBatch && maxIter > 0) {
    uint256 batchShares = _withdrawalBatches[batch].totalShares;
    if (batchShares >= shares) {
        _withdrawalBatches[batch].totalShares -= shares;
        break;
    } else {
        _withdrawalBatches[batch].totalShares = 0;
        shares -= batchShares;
        batch++;
    }
    --maxIter;
}
```

If a user cancels a request that was created in a later batch, this logic subtracts the cancelled shares from the earliest batch's `totalShares`. That can make `pendingBatch.totalShares` smaller than the actual FIFO-queued shares at the head of the Vault queue.

During processing, the strategy limits processing by `min(queueMaxShares, batch.totalShares)` and stops once it reaches the next (not-yet-expired) batch.

Impact: Queued withdrawals that should be eligible (earlier batch already expired and liquidity exists) can be artificially delayed until a later batch expires, because the strategy advances to a future batch while some earlier-batch shares still remain in the queue head. This can degrade withdrawal liveness and create unexpected waiting periods for lenders.

Proof of Concept: Add the following test to `test/strategies/AccountableOpenTermBatch.t.sol`:

```
function test_cancelFromFutureBatch_canDelayEarlierBatchProcessing() public {
    _setupLoanWithBatches(4 days, 7 days);

    // Three lenders deposit, borrower borrows everything (no immediate liquidity).
    vm.prank(alice);
    usdcOpenTermVault.deposit(USDC_AMOUNT, alice, alice);
    vm.prank(bob);
    usdcOpenTermVault.deposit(USDC_AMOUNT, bob, bob);
    vm.prank(charlie);
    usdcOpenTermVault.deposit(USDC_AMOUNT, charlie, charlie);

    vm.prank(borrower);
    usdcOpenTermLoan.borrow(USDC_AMOUNT * 3);

    // Batch 0: Alice + Bob queue withdrawals in the first interval.
    uint256 aliceB0 = USDC_AMOUNT / 2;
    uint256 bobB0 = USDC_AMOUNT / 3;
    vm.prank(alice);
    usdcOpenTermVault.requestRedeem(aliceB0, alice, alice);
    vm.prank(bob);
    usdcOpenTermVault.requestRedeem(bobB0, bob, bob);

    WithdrawalBatch memory b0Before = usdcOpenTermLoan.withdrawalBatches(0);
    assertEq(b0Before.totalShares, aliceB0 + bobB0, "batch0 tracks Alice+Bob");

    // Move to next interval -> Batch 1 is created by Charlie.
    vm.warp(block.timestamp + 7 days);
    uint256 charlieB1 = USDC_AMOUNT / 5;
    vm.prank(charlie);
    usdcOpenTermVault.requestRedeem(charlieB1, charlie, charlie);

    assertEq(usdcOpenTermLoan.currentBatch(), 1, "batch1 created");

    WithdrawalBatch memory b1Before = usdcOpenTermLoan.withdrawalBatches(1);
    assertEq(b1Before.totalShares, charlieB1, "batch1 tracks Charlie");

    // Charlie cancels. Strategy reduces starting from pendingBatch (0),
    // even though Charlie's request was created in batch 1.
    vm.prank(charlie);
    usdcOpenTermVault.cancelRedeemRequest(charlie, charlie);

    WithdrawalBatch memory b0AfterCancel = usdcOpenTermLoan.withdrawalBatches(0);
    WithdrawalBatch memory b1AfterCancel = usdcOpenTermLoan.withdrawalBatches(1);

    // NOTE: This shows the core accounting problem: batch0 shrinks (even though Alice+Bob are still
    ↪  queued),
    // and batch1 stays unchanged (even though Charlie is no longer queued).
    assertEq(
        b0AfterCancel.totalShares,
        (aliceB0 + bobB0) - charlieB1,
        "batch0 reduced by Charlie cancel (mis-attributed)"
    );
    assertEq(b1AfterCancel.totalShares, charlieB1, "batch1 unchanged (stale metadata)");
```

```
// Queue now contains only Alice+Bob.
assertEq(usdcOpenTermVault.totalQueuedShares(), aliceB0 + bobB0, "queue excludes cancelled
↪  Charlie");

// We are already past batch0 expiry (4d) and before batch1 expiry (7d+4d).
assertGe(block.timestamp, b0Before.expiry, "past batch0 expiry");
assertLt(block.timestamp, b1Before.expiry, "before batch1 expiry");

// Borrower repays enough liquidity to process ALL queued shares (Alice+Bob).
// Due to understated batch0.totalShares, processing only does (alice+bob-charlie) shares and then
↪  stops at batch1.
usdc.mint(borrower, USDC_AMOUNT * 3);
vm.startPrank(borrower);
usdc.approve(address(usdcOpenTermLoan), type(uint256).max);
usdcOpenTermLoan.repay(USDC_AMOUNT * 3);
vm.stopPrank();

// Remaining queue shares == the "missing" amount (charlieB1), even though Charlie cancelled.
// These are actually part of Alice/Bob's earlier requests that got pushed into the next batch
↪  window.
assertEq(
    usdcOpenTermVault.totalQueuedShares(),
    charlieB1,
    "earlier requests rolled into next batch window (delayed until batch1 expiry)"
);
assertEq(usdcOpenTermLoan.pendingBatch(), 1, "pendingBatch advanced to batch1 and now blocks further
↪  processing");

// Alice requested 500e11 shares and should be fully claimable:
uint256 aliceClaimableShares = usdcOpenTermVault.maxRedeem(alice);
assertEq(aliceClaimableShares, 500_000_000_000, "Alice fully claimable in batch0");

// Bob requested 333333333333 shares, but only part of it was processed due to the bug.
// From the trace: bob got RedeemClaimable(..., shares: 133333333333)
uint256 bobClaimableShares = usdcOpenTermVault.maxRedeem(bob);
assertEq(bobClaimableShares, 133_333_333_333, "Bob only partially claimable in batch0 (bug)");

// The remainder should still be queued (333333333333 - 133333333333 = 200000000000)
assertEq(usdcOpenTermVault.totalQueuedShares(), 200_000_000_000, "Remaining Bob shares still
↪  queued");

// Demonstrate users can redeem what is currently claimable:
vm.prank(alice);
usdcOpenTermVault.redeem(aliceClaimableShares, alice, alice);

vm.prank(bob);
usdcOpenTermVault.redeem(bobClaimableShares, bob, bob);

// After redeeming claimable amounts, queue should still contain Bob's remainder
assertEq(usdcOpenTermVault.totalQueuedShares(), 200_000_000_000, "Bob remainder still queued after
↪  partial redeem");

// Remaining shares can't become claimable until batch1 expires.
// Warp past batch1 expiry and trigger processing again.
WithdrawalBatch memory batch1 = usdcOpenTermLoan.withdrawalBatches(1);
vm.warp(batch1.expiry + 1);
usdcOpenTermLoan.processAvailableWithdrawals();

// Now Bob's remainder should become claimable:
uint256 bobClaimableAfter = usdcOpenTermVault.maxRedeem(bob);
assertEq(bobClaimableAfter, 200_000_000_000, "Bob remainder becomes claimable only after batch1
↪  expiry");
```

```
    // And Bob can finally redeem the rest:
    vm.prank(bob);
    usdcOpenTermVault.redeem(bobClaimableAfter, bob, bob);

    assertEq(usdcOpenTermVault.totalQueuedShares(), 0, "Queue fully drained after delayed processing");
}
```

Recommended Mitigation: Ensure cancellations decrement the correct batch. Track the batch id for each redeem request (or controller's pending request) at queue time and subtract from that batch on cancel.

Accountable: Fixed in commit ec9ec5e

Cyfrin: Verified. Cancellation now subtracts shares from the correct batch(es) via per-controller batch tracking.

7.1.5 Missing modifiers on `YieldStrategyFactory.createYieldStrategy` can lead to deployment of unverified strategies

Description: Function `createYieldStrategy` enables permissionless deployment of `AccountableYield` strategies. However, during the deployment process, the function does not:

1. Verify the paused status using the `whenNotPaused` modifier
2. Verify the transaction authentication data using `onlyVerified` (if a signer is set)
3. Verify whether the asset is whitelisted using `onlyWhitelistedAsset`

Proof of Concept: As we can observe, other strategy factories such as `OpenTermFactory` and `FixedTermFactory` implement this verification.

YieldStrategyFactory.sol

```
function createYieldStrategy(YieldFactoryParams memory params)
        external
        returns (address strategyProxy, address vault)
    {
```

OpenTermFactory.sol

```
function createOpenTermLoan(OpenTermFactoryParams memory params)
        external
        whenNotPaused
        onlyVerified
        onlyWhitelistedAsset(params.asset)
        returns (address strategyProxy, address vault)
    {
```

FixedTermFactory.sol

```
function createFixedTermLoan(FixedTermFactoryParams memory params)
        external
        whenNotPaused
        onlyVerified
        onlyWhitelistedAsset(params.asset)
        returns (address strategyProxy, address vault)
    {
```

Recommended Mitigation: Consider applying the `whenNotPaused`, `onlyVerified` and `onlyWhitelistedAsset` modifiers on the function `createYieldStrategy`.

Accountable: Fixed in commit f8d4a3f

Cyfrin: Verified. Modifiers now applied.

7.2 Low Risk

7.2.1 `AccountableOpenTerm` rate publish/rollback does not refresh delinquency status

Description: In `AccountableOpenTerm`, the new rate update flows (`publishRate()` / `rollbackRate()`) call `_accrueInterest()` and then update core economic parameters (e.g., `interestRate`, scale-factor–related state, and accrual timestamps). However, these functions do not call `_updateDelinquentStatus()` after mutating the loan's economic state.

Impact: Delinquency status can remain stale until a later interaction triggers `_updateDelinquentStatus()`. This can cause temporary inconsistencies in delinquency tracking and penalty timing, and may affect monitoring/automation that relies on delinquency state immediately after rate changes.

Recommended Mitigation: Call `_updateDelinquentStatus()` at the end of both `publishRate()` and `rollbackRate()` so delinquency state always reflects the latest rate/accrual state.

Accountable: Fixed in commits 39c60c7 and f350a8d.

Cyfrin: Verified. `_updateDelinquentStatus()` now called from both `publishRate()` and `rollbackRate()`.

7.2.2 Fee structure updates can trigger accrual after loan has ended

Description: In both `AccountableYield` and `AccountableOpenTerm`, `onFeeStructureChange()` only checks `if (_loan.startTime != 0)` before accruing:

```
if (_loan.startTime != 0) {
    _accrueFees();     // AccountableYield
    // or _accrueInterest(); // AccountableOpenTerm
}
```

Since `startTime` is set once and not reset when a loan is `Repaid` or in default, fee-structure changes can still trigger accrual logic after the loan is no longer ongoing.

Impact: In `AccountableYield`, management fees are time-based; if no accrual happens after repayment, a later fee-structure update can "catch up" and mint a large amount of fee shares for the elapsed time since the last fee accrual, diluting holders unexpectedly. In `AccountableOpenTerm`, the hook can similarly update interest bookkeeping (or potentially revert if accrual assumes an ongoing loan). However this requires the protocol to update the fee structure after the loan has ended, which is unlikely.

Recommended Mitigation: Gate `onFeeStructureChange()` by loan state (e.g., only accrue when the loan is ongoing), rather than `startTime != 0`. For example, require `loanState == Ongoing*` before calling `_accrueFees()` / `_accrueInterest()`, or use `_requireLoanOngoing()` similar to other calls.

Accountable: Fixed in commit 5f8fd3

Cyfrin: Verified. Check changed to `loanState == LoanState.OngoingDynamic`.

7.2.3 Delinquency status update in `AccountableOpenTerm` hooks uses pre-queue state

Description: The Vault calls strategy hooks (e.g., `AccountableStrategy::onRequestRedeem`, `onDeposit`, `onMint`) before the Vault updates the state that these actions affect (queue totals for redeem requests, and `totalAssets`/liquidity for deposits/mints).

However, `AccountableOpenTerm` updates delinquency status inside the hooks.

As a result, delinquency calculations that depend on Vault-side values (e.g., queued shares / available liquidity derived from `totalAssets`, `reservedLiquidity`, `totalQueuedShares`, etc.) can be evaluated using a pre-action snapshot:

- `AccountableOpenTerm::onRequestRedeem`: for queued (non-instant) requests, the request is only enqueued after the hook returns, so delinquency is checked before the new queued shares are reflected.
- `AccountableOpenTerm::onDeposit` / `onMint`: the hook runs before the Vault receives assets / updates totals, so delinquency can be checked before the new liquidity from the deposit/mint is reflected.

Impact: Delinquency status may lag by one interaction (or until another status update is triggered). For example:

- a queued redeem may not immediately mark the loan delinquent, and/or
- a deposit/mint that would restore liquidity may not immediately clear delinquency.

This is primarily a correctness / timing issue unless delinquency gating is expected to be exact within the same transaction.

Recommended Mitigation: Consider passing the changes in shares and assets to the delinquency calculation so that it can account for the added/removed shares/assets. Or use the same pattern as `Vault::cancelRedeemRequest` where the. `strategy.updateLateStatus()` hook is called at the end.

Accountable: Fixed in commit `5f815ee`

Cyfrin: Verified. Delinquency update removed from the strategy hooks and each vault function now calls `trategy.updateLateStatus`.

7.2.4 Increasing `AccountableOpenTerm.loan.withdrawalPeriod` from `0` can cause withdrawals to become stuck

Description: When `LoanTerms.withdrawalPeriod == 0`, `AccountableOpenTerm::_createOrAddWithdrawalBatch` returns immediately and does not create/update any batch metadata.

However, users can still end up queued (e.g., insufficient liquidity → `requestRedeem()` is not immediately fulfillable). If terms are later updated to a non-zero `withdrawalPeriod`, `_processAvailableWithdrawals()` switches into "batch mode" and processes shares bounded by `WithdrawalBatch.totalShares`, meaning queued shares that were accumulated while `withdrawalPeriod == 0` may have no corresponding batch totals to drive processing.

Impact: Queued withdrawals created while `withdrawalPeriod == 0` can become stuck or perceived as stuck after switching to `withdrawalPeriod > 0`, because batch-mode processing depends on batch metadata that was never created for those queued shares. This can lead to delayed withdrawals and operational term toggling to recover.

Recommended mitigation: Either don't allow increases of the `withdrawalPeriod` when there's still queued shares or:

When transitioning from `withdrawalPeriod == 0` → `withdrawalPeriod > 0`, ensure queued withdrawals are materialized into batch metadata. Options include:

- In `acceptTerms()` (or the terms-activation path), if the *new* `withdrawalPeriod > 0` and `totalQueuedShares() > 0`, **initialize/seed** the current batch with `totalShares = totalQueuedShares()`, with appropriate `startTime/expiry` alignment.
- Alternatively, adjust `_processAvailableWithdrawals()` so that if `withdrawalPeriod > 0` but batch metadata is missing/empty while the queue is non-empty, it either:
  - falls back to the "zero-period" processing path once, or
  - auto-creates a batch reflecting the current queued amount before processing.

Accountable: Fixed in commit `10396d4`

Cyfrin: Verified. `acceptTerms` now reverts if the withdrawal period is increased from 0 and there's still queued shares.

7.2.5 Function `execute` overwrites seenSigner values irrespective of request age

Description: Function `publish` allows authenticated signers to publish requests for the current batch in process. By design, signers are allowed publish multiple requests in a batch, with each request having its own distinct timestamp.

When an authorized executor calls the `execute` function, this for loop will overwrite the signer's first request value with the second request value (assuming only two requests have been published). However, it is possible that the `request.timestamp` of the second request is older than the timestamp of the first request. In this case,

the function uses the seen signer's relatively older request value instead of the latest, which can lead to slightly inaccurate publish rates.

```
        for (uint256 j = 0; j < uniqueSigners; ++j) {
            if (requests[i].signer == seenSigners[j]) {
                // Update to latest value from this signer
                values[j] = requests[i].value;
                isDuplicate = true;
                break;
            }
        }
```

Proof of Concept: Let's take a simple example:

- Alice submits two requests - R1 and R2
- The timestamps of R1 and R2 are 20 and 10 respectively.
- During execution, R1's value is overwritten by R2's value even though R2 is a relatively older request.

Recommended Mitigation: If a signer has multiple requests in a batch, ensure the value is not overwritten unless the timestamp of the request is fresher.

Accountable: Fixed in commit 141ca3b

Cyfrin: Verified. Only updates if timestamp is later.

7.2.6 `lastTotalAssets` stores stale value due to update before penalty accrual

Description: Across the `AccountableYield` strategy, variable `lastTotalAssets` stores the value returned from function `_totalAssets`. Since `accruedPenalties` is a factor taken into consideration in `_totalAssets`, the code should ensure penalties are accrued before updating the `lastTotalAssets`.

```
/// @dev Total assets managed = vault assets + deployed assets + accrued penalties
    function _totalAssets(address vault_) internal view returns (uint256) {
        return IAccountableVault(vault_).totalAssets() + deployedAssets + accruedPenalties;
    }
```

However, in all instances across the `AccountableYield` contract, `lastTotalAssets` does not accrue penalties before.

`AccountableYield.borrow`

`AccountableYield.onDeposit`

`AccountableYield.onMint`

`AccountableYield._accrueFees/_accruedFeeShares`

Similar instances also exist when `_totalAssets` is accessed directly before accruing penalties:

`AccountableYield._accruedFeeShares` - In this particular instance, function `_accruedFeeShares` also returns this `newTotalAssets` value to function `_sharePrice`, which leads to a stale share price.

`AccountableYield._calculateRequiredLiquidity`

Recommended Mitigation: Consider accruing penalties before updating `lastTotalAssets` as well as before directly accessing the value returned from function `_totalAssets`.

Accountable: Fixed in commit 97f8b1a

Cyfrin: Verified. Penalties now accrued in the above stated cases.

7.3 Informational

7.3.1 `AccountableOpenTerm` manual interest rate proposal is unbounded

Description: The manual interest rate path can propose/queue an interest rate without enforcing an upper bound. By contrast, the DVN publishing path enforces a cap when applying rates:

- DVN flow: `AccountableOpenTerm::publishRate(uint256 newRate)` checks `newRate` against `MAX_PUBLISH_RATE` before applying it.
- Manual flow: `AccountableOpenTerm::proposeInterestRate(...)` queues a pending rate, and `approveInterestRateChange()` applies it, but the queued rate is not capped.

Recommended Mitigation: Add the same bounds check in `proposeInterestRate(...)` (preferred), so invalid/extreme rates cannot be queued.

Accountable: Fixed in commit `4a737a6`

Cyfrin: Verified.

7.3.2 `AccountableYield::setNavGracePeriod` uses `Unauthorized` error for invalid input

Description: `AccountableYield.setNavGracePeriod()` reverts with `Unauthorized()` when `period < MIN_NAV_GRACE_PERIOD`, even though this is an input validation failure rather than an access control issue.

Consider using a dedicated error for invalid parameters (e.g., `InvalidNavGracePeriod()`), or reuse a generic input-validation error.

Accountable: Fixed in commit `d051169`

Cyfrin: Verified.

7.3.3 `DVNPublisher::publish` does not enforce a maximum age for updates

Description: `DVNPublisher::publish` validates that `request.timestamp` is not in the future, but it does not enforce a maximum age (i.e., it does not reject requests that are already stale at submission time). Staleness is only handled later during `DVNPublisher::execute`.

Consider adding a check in `publish()` to reject already-stale requests, e.g. `require(request.timestamp + maxStaleness >= block.timestamp)`, to reduce clutter in `_pendingRequests`.

Accountable: Fixed in commit `5afc5fd`

Cyfrin: Verified.

7.3.4 Consider reverting in `publishedDataByBatchId` for invalid batch IDs

Description: Function `publishedDataByBatchId` returns published data however it does not ensure the `id` parameter is less than the `currentBatchId`. Due to this, the function will still return an empty PublishedData struct for invalid IDs. While this poses no immediate risk, it is safer to reject invalid ID values to avoid issues in the future with integrations.

```
function publishedDataByBatchId(uint256 id) external view returns (PublishedData memory) {
        return _publishedData[id];
    }
```

Recommended Mitigation: Consider reverting if `id` is greater than or equal to the `currentBatchId`.

Accountable: Fixed in commit `d721846`

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Precompute `baseSlot` in `AtomicBatcher::_getNonceSlot`

Description: `AtomicBatcher::_getNonceSlot` computes the ERC-7201 namespace base slot at runtime each time it is called:

```
function _getNonceSlot(address account) private pure returns (bytes32) {
    bytes32 namespaceHash = keccak256(bytes(_NAMESPACE));
    bytes32 baseSlot = keccak256(abi.encode(uint256(namespaceHash) - 1)) & ~bytes32(uint256(0xff)); //
    ↪  @audit can be pre-computed
    return keccak256(abi.encode(account, uint256(baseSlot)));
}
```

Since the namespace is constant, the derived `baseSlot` can be precomputed and stored as a `bytes32` constant:

```
// keccak256(abi.encode(uint256(keccak256("accountable.atomicbatcher.nonce.v1")) - 1)) &
↪  ~bytes32(uint256(0xff))
bytes32 private constant NONCE_BASE_SLOT =
↪  0xa68386067ee8ee669468449acf0ad3e2ae0d09e4d99f78eaa329c6681c06b900;
```

Accountable: Fixed in commit `fce94d2`

Cyfrin: Verified.

7.4.2 Precompute `callTypeHash` in `AtomicBatcher::_hashCallArray`

Description: `AtomicBatcher::_hashCallArray` recomputes the call type hash (the `keccak256` of the call-type string / type description) on every invocation:

```
function _hashCallArray(Call[] calldata calls) private pure returns (bytes32) {
    bytes32 callTypeHash = keccak256("Call(address target,uint256 value,bytes data)");
```

Since this value is constant, it can be precomputed once as a `bytes32` constant.

Consider defining `bytes32 private constant _CALL_TYPEHASH = keccak256("...");` and use `_CALL_TYPEHASH` in `_hashCallArray()` instead of computing it each time.

Accountable: Fixed in commit `6a63afe`

Cyfrin: Verified.

7.4.3 Reuse `fm` Instead of re-instantiating `IFeeManager` in `AccountableOpenTerm::_mintFeeShares`

Description: In `AccountableOpenTerm::_mintFeeShares`, the treasury address is fetched by re-casting `feeManager`:

```
address treasury_ = IFeeManager(feeManager).treasury();
```

However, the `IFeeManager fm` interface is already passed into the function, so this extra cast/load is unnecessary:

```
address treasury_ = fm.treasury();
```

Accountable: Fixed in commit `5e13285`

Cyfrin: Verified.

7.4.4 Reuse `aum_` in `_accrueFeeShares` to avoid recomputing debt

Description: In `AccountableOpenTerm::_accrueFeeShares`, debt is recomputed as:

```
uint256 debt = _loan.outstandingPrincipal.mulDiv(scaleFactor_, PRECISION);
```

However, the same debt/AUM value is already computed in `_accrueInterest()` and passed down as `aum_`. This makes the multiplication/division redundant and also keeps `scaleFactor_` as an unnecessary parameter.

Consider using `aum_` directly in `_accrueFeeShares` (e.g., `uint256 debt = aum_;`) and remove the `scaleFactor_` parameter from the function signature and call sites to save gas and simplify the code.

Accountable: Fixed in commit `f350a8d`

Cyfrin: Verified.

7.4.5 Only update `deployedAssets` when `remaining > 0` in `AccountableYield::repay`

Description: In `AccountableYield::repay`, `deployedAssets` is read and conditionally reduced unconditionally:

```
uint256 deployed = deployedAssets;
deployedAssets -= Math.min(remaining, deployed);
```

However, this has no effect when `remaining == 0` (since `Math.min(0, deployed) == 0`). You already branch on `remaining > 0` immediately after for principal reduction:

```
// Reduce outstanding principal
if (remaining > 0) {
     uint256 outstanding = _loan.outstandingPrincipal;
    _loan.outstandingPrincipal = outstanding > remaining ? outstanding - remaining : 0;
}
```

Consider reducing `deployedAssets` inside the existing `if (remaining > 0)` block to avoid an unnecessary storage read and write when there is no remaining repayment amount:

```
// Reduce deployedAssets and outstanding principal
if (remaining > 0) {
    // Assets are moving from external → vault
    uint256 deployed = deployedAssets;
    deployedAssets -= Math.min(remaining, deployed);

     uint256 outstanding = _loan.outstandingPrincipal;
    _loan.outstandingPrincipal = outstanding > remaining ? outstanding - remaining : 0;
}
```

Accountable: Fixed in commit `eec49ac`

Cyfrin: Verified.

7.4.6 Consider removing redundant zero address check from `createYieldStrategy`

Description: Function `createYieldStrategy` deploys instances of the `AccountableYield` strategy using the `Create2` library's `deploy` function. After deployment, it ensures the `strategyProxy` is not `address(0)`.

```
strategyProxy = Create2.deploy(0, params.salt, strategyProxyBytecode);
if (strategyProxy == address(0)) revert FailedDeployment(ZERO_LOAN_PROXY_ADDRESS);
```

However, this is not required since the `deploy` function already checks for this and reverts early.

```
function deploy(uint256 amount, bytes32 salt, bytes memory bytecode) internal returns (address addr) {
        if (address(this).balance < amount) {
            revert Create2InsufficientBalance(address(this).balance, amount);
        }
        if (bytecode.length == 0) {
            revert Create2EmptyBytecode();
        }
        /// @solidity memory-safe-assembly
        assembly {
            addr := create2(amount, add(bytecode, 0x20), mload(bytecode), salt)
```

```
        }
        if (addr == address(0)) {
            revert Create2FailedDeployment();
        }
    }
```

Recommended Mitigation: Consider removing the zero address check

```
- if (strategyProxy == address(0)) revert FailedDeployment(ZERO_LOAN_PROXY_ADDRESS);
```

Accountable: Fixed in commit ec3b6b9

Cyfrin: Verified. Optimization also done for Open- and FixedTerm factories.

7.4.7 Consider emitting events early to save gas

Description: Function `DVNPublisherFactory.setImplementation` can emit event `ImplementationSet` before the `implementation` state change to avoid creating and accessing an unnecessary memory variable.

DVNPublisherFactory.sol#L34

```
function setImplementation(address implementation_) external onlyOwner {
        address oldImplementation = implementation;
        implementation = implementation_;
        emit ImplementationSet(oldImplementation, implementation_);
    }
```

Similar instances exist in the `DVNPublisher` and `AccountableYield` contract:

DVNPublisher.sol#L125

```
function setThreshold(uint256 threshold_) external onlyManager {
        if (threshold_ > MAX_THRESHOLD || threshold_ == 0) revert InvalidThreshold();

        uint256 oldThreshold = threshold;
        threshold = threshold_;

        emit ThresholdSet(oldThreshold, threshold_);
    }
```

DVNPublisher.sol#L153

```
function setMaxStaleness(uint256 maxStaleness_) external onlyManager {
        if (maxStaleness_ == 0) revert ZeroValue();

        uint256 oldMaxStaleness = maxStaleness;
        maxStaleness = maxStaleness_;

        emit MaxStalenessSet(oldMaxStaleness, maxStaleness_);
    }
```

DVNPublisher.sol#L163

```
    /// @inheritdoc IDVNPublisher
    function setMaxDeviation(uint256 maxDeviation_) external onlyManager {
        uint256 oldMaxDeviation = maxDeviation;
        maxDeviation = maxDeviation_;

        emit MaxDeviationSet(oldMaxDeviation, maxDeviation_);
    }
```

AccountableYield.sol#L226

```
function setNavGracePeriod(uint256 period) external onlyManager {
        if (period < MIN_NAV_GRACE_PERIOD) revert Unauthorized();

        uint256 oldPeriod = navGracePeriod;
        navGracePeriod = period;

        emit NavGracePeriodSet(oldPeriod, period);
    }
```

Recommended Mitigation: Consider emitting the event early before the state update.

Accountable: Fixed in commit 1a7ce24

Cyfrin: Verified.