Security Assessment Report

Catalysis

March 2026

Prepared for Catalysis

certora catalysis

Table of Contents

certora catalysis

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Catalysis	https://github.com/0xcatalysis/coverage/tree/dhruv/coveredvaultwrapper	f62df10 (initial commit) a2945e0 (fix commit)	Solidity

Project Overview

This document describes the security review of **Catalysis**. The work was undertaken from **March 23rd, 2026** to **March 31st, 2026**.

The following contract list is included in our scope:

`src/defi/*`

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

The system is built around CoveredVaultWrapper, a user-facing vault that provides per-user insurance on top of deposits routed into a configured Morpho V2 vault.

Users deposit the underlying ERC-20 asset into CoveredVaultWrapper, an upgradeable ERC-4626 vault whose shares are non-transferable, so the insured cost basis cannot be moved between accounts. The wrapper then deposits those assets into a configured Morpho V2 vault, and the wrapper holds the Morpho shares, while it tracks each user's internal entitlement to those Morpho shares (userMorphoShares) and their cost basis principal (userPrincipal).

Insurance is enforced at withdrawal time. When a user redeems wrapper shares, the wrapper calculates the portion of the user's principal attributable to the shares being burned, redeems

the corresponding amount of Morpho shares back into assets, deducts any premium owed, and then checks whether the user is underwater relative to their insured basis. If the policy is active and there is a shortfall (principal basis greater than net assets received), the wrapper files an insurance claim via the ClaimManager and transfers any payout directly to the withdrawing user alongside their net redemption proceeds.

Premiums are not baked into the share price as yield. Instead, the wrapper accrues premiums over time on total principal using a reward-per-token style accumulator (cumulativePremiumPerUnit), tracking each user's premium debt (userPremiumDebt). Collected premium is accumulated as pendingPremium and later transferred to a designated premiumRecipient via collectPremium(). The insurance policy is bound once via onPolicyBound(), which validates the policy parameters and configures a global deposit cap derived from the coverage limit and maxCoverableLossBps.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

certora catalysis

Threat and Security Overview

The CoveredVaultWrapper is a user-facing ERC-4626 vault that routes deposits into a configured Morpho V2 vault and adds per-user insurance and premium accounting on top.

The contract trusts (1) the Morpho vault to correctly represent value via convertToAssets() / redeem(), and (2) the ClaimManager / PolicyManager to correctly enforce policy terms and pay out claims. During the review, the focus is therefore on ways a user can tamper with accounting and timing to (i) claim more insurance than intended, (ii) avoid paying premiums, (iii) force unfair distribution of limited coverage, or (iv) cause denial-of-service / stuck withdrawals via Morpho configuration (gates/adapters) and the wrapper's fallback logic.

Key takeaways:

- **Insurance can be driven by Morpho fee dilution, not just "real losses".** If the Morpho vault charges non-zero management/performance fees (implemented via share dilution), a user's Morpho redemption value can drop even without any adverse market event. Since the wrapper treats "principal vs redeemed assets" as loss, this fee-driven dilution can appear as shortfall and may trigger insurance claims, effectively subsidizing Morpho fees with insurance cover, though this edge would be handled by applying `deductibleBps` to the basis and losses at or below this threshold would yield a $0 payout.

- **Timing and policy lifecycle matter more than before.** Premium accrual and insurance eligibility depend on policy activation windows (`startTime`, maturity, and `gateGracePeriod`) while deposits may occur outside the active window depending on configuration. Attack surfaces include depositing shortly before activation, withdrawing immediately when the policy becomes active, and exploiting how `lastPremiumAccrualTime` and `_activeElapsedFrom()` define which time intervals are billable.

- **ERC-4626 compliance pitfalls can become real constraints.** The wrapper overrides `redeem()` to return the actual transferred amount, but still relies on ERC-4626 max functions for `withdraw()`. If view functions like `maxWithdraw()` are conservative but incorrect (e.g., double-counting premium), it can cause on-chain reverts in OpenZeppelin's ERC-4626 `withdraw()` path, meaning users would be unable to withdraw amounts that are otherwise redeemable via `redeem()`.

Test coverage notes:

- Since the wrapper integrates tightly with Morpho's interest accrual, fee minting, and gating behavior, meaningful tests should include end-to-end scenarios with a real Morpho vault configuration (including non-zero fees and gate toggles). Unit tests that bypass Morpho internals (or rely only on mocked values) risk missing timing and accounting edge cases that only appear when Morpho's `accrueInterest()` / `accrueInterestView()` paths are exercised.

certora catalysis

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	1	1	1
Medium	4	4	4
Low	3	3	1
Informational	-	-	-
Total	8	8	6

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora catalysis

Detailed Findings

ID	Title	Severity	Status
H-01	Premium accrual system will lose premiums due to rounding in 6-decimal tokens	High	Fixed
M-01	depositCap can't be lowered without affecting the maxCoverableLossBps which may be undesirable in situations where the underlying Morpho vault is at a loss	Medium	Fixed
M-02	User Can Get Insurance Without Paying Any Premiums	Medium	Fixed
M-03	`maxWithdraw()` double counts premium and understates withdrawable assets	Medium	Fixed
M-04	Premium accrues for time before startTime due to missing clamp	Medium	Fixed
L-01	lastKnownMorphoValue staleness creates an outdated _liabilityHeadroom value	Low	Acknowledged
L-02	Multiple instances of rounding directions in favor of the user	Low	Fixed
L-03	totalAssets() can theoretically return 0 with outstanding shares, which would allow users to mint a disproportionate amount of shares	Low	Acknowledged

High Severity Issues

H-01: Premium accrual system will lose premiums due to rounding in 6-decimal tokens

Severity: High	Impact: High	Likelihood: Medium
Files: CoveredVaultAccrual.sol	Status: Fixed	

Description:

In certain cases, mainly due to low totalPrincipal and shorter activeElapsed, the premium calculation formula rounds to zero for 6-decimal tokens (e.g. USDC) due to an 18-decimal divison, the accrued premium can be unintentionally lost due to it or this can be weaponized via repeated calls to avoid premium and skew totalAssets().

During every deposit and withdraw call, as well as any time that a calculation involves totalAssets, premiums are accrued based on the elapsed time, as well as the premiumRatePerSecond. This is done via _accruePremium, and _accruedPremiumFor.

The accrued premium is calculated in the following way:

```
uint256 accrued = _totalPrincipal > 0
            ? _totalPrincipal.mulDiv(premiumRatePerSecond * activeElapsed, WAD,
Math.Rounding.Floor)
            : 0;
```

As it can be seen from above, after multiplying the total principal with the multiplication result of the elapsed time and premiumRatePerSecon, it's divided with WAD - the problem is that WAD is 1e18, and the calculation is intended for 18-decimal tokens. This results in premium losses for 6-decimal tokens, due to rounding it down to 0. The effects are amplified during lower totalPrincipal and shorter elapsed times.

Recommendations:

Calculations can be adjusted for 6-decimal tokens to be "converted" to 18-decimals before performing any accrual calculations to make sure that there are no losses or use a virtual share calculation approach.

Customer Response:

Fixed in a2945e0.

Fix Review:

Fix confirmed.

Medium Severity Issues

M-01: depositCap can't be lowered without affecting the maxCoverableLossBps which may be undesirable in situations where the underlying Morpho vault is at a loss

Severity: Medium	Impact: High	Likelihood: Low
Files: CoveredVaultWrapper.sol CoveredVaultAdmin.sol	Status: Fixed	

Description:

The problem is that depositCap can't be increased/decreased in a stand-alone, isolated matter, but it's always done programmatically through maxCoverableLossBps changes. This approach can present a problem in certain situations.

```
depositCap = (_coverageLimit * BPS_DENOMINATOR) / uint256(maxCoverableLossBps);
```

Currently there is no way to increase or decrease the deposit cap for risk management without also retroactively changing the payout percentage that existing depositors rely on. This means that in situations in which the underlying vault is at a loss, we would have to increase the payout percentage that current depositors will get in order to limit deposits.

This also represents a problem in the opposite scenario, in order to increase the depositCap, current depositors' max coverable loss would have to be decreased.

Having to decrease the maxCoverableLossBps of an already stressed vault, would also invite new deposits into the vault, due to an increasing depositCap.

Recommendations:

Decouple the deposit cap from the maxCoverableLossBps, and allow it to be adjusted independently by an admin.

Customer Response:

Fixed in a2945e0.

Fix Review:

Fix confirmed.

M-02: User Can Get Insurance Without Paying Any Premiums

Severity: Medium	Impact: High	Likelihood: Low
Files: CoveredVaultWrapper.sol	Status: Fixed	

Description:

Premium only accrues while the policy is active, but insurance becomes available immediately once block.timestamp >= startTime. Because _accruePremium() always updates lastPremiumAccrualTime (even when no premium is accrued), a user can keep their premium checkpoint "fresh" right before startTime and then claim insurance for a large shortfall immediately after startTime while paying almost no premium.

Consider the following scenario:

Policy is bound at Day 0 , but the policy's startTime is Day 30.

1) After onPolicyBound(), attacker deposits a large amount, e.g. 10,000,000 assets, at Day 0 into the wrapper.

2) This sets up a large userPrincipal i.e. userPrincipal[attacker] = 10,000,000, but since the policy is not active yet, premium accrual is effectively not charging for this time window.

3) Between Day 0 → Day 30, the Morpho vault suffers a loss so the attacker's position is worth, say, 8,000,000 assets by Day 30.

4) Right before startTime (just a block prior to the startTime block), attacker makes a tiny additional deposit (just enough to mint non-zero wrapper shares).

5) This call triggers _accruePremium(), and even though the policy is still inactive (so activeElapsed = 0), the function updates lastPremiumAccrualTime to one block before startTime.

6) At block.timestamp == startTime (Day 30) or immediately after, attacker redeems/withdraws._isPolicyActive() is now true (because block.timestamp >= startTime), so insurance is eligible.

7) But the premium charged since the last checkpoint is negligible (~1 block worth), because lastPremiumAccrualTime was refreshed right before start.The vault computes a large shortfall

(principal vs assets received) and the attacker can receive an insurance payout immediately, while paying almost no premium.

Recommendations:

Make sure onPolicyBound() is invoked at or close to the startTime.

Customer Response:

Fixed in a2945e0.

Fix Review:

Fix confirmed.

M-03: `maxWithdraw()` double counts premium and understates withdrawable assets

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: CoveredVaultWrapper.sol	Status: Fixed	

Description:

maxWithdraw() can return a value that is too low because it subtracts premium twice:

once via _convertToAssets(shares) (which is based on totalAssets() which accounts for premiums) ->

```
function totalAssets() public view override (ERC4626Upgradeable, IERC4626)
returns (uint256) {
        uint256 mv = _morphoValue();
        uint256 totalLiability = totalPremiumLiability + _accruedPremiumFor(mv);
        return mv > totalLiability ? mv - totalLiability : 0;
    }
```

and again explicitly by subtracting the user's owedPremium (and pending premium delta).

Consider the following scenario:

1) Alice deposits 100 assets. Morpho value stays 100.

2) Over time, premium accrues: totalPremiumLiability = 10 (no pending delta for simplicity).

3) Now:totalAssets() = 100 - 10 = 90_convertToAssets(AliceShares) returns 90maxWithdraw() then computes Alice's owedPremium = 10 and subtracts it again:maxWithdraw = 90 - 10 = 80

So maxWithdraw() reports 80, even though the share-to-asset conversion already reflected the 10 premium liability when it returned 90.

Recommendations:

Compute maxWithdraw() using a single premium basis, not both/

Customer Response:

Fixed in a2945e0 and ef852b3.

Fix Review: Previous vulnerability was fixed, the new fix intentionally returns the conservative maxWithdraw value. Which is a safe lower bound tied to the ERC-4626 share-price model. Natspec updates in this ef852b3 commit reflect this.

M-04: Premium accrues for time before startTime due to missing clamp

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: CoveredVaultWrapper.sol	Status: Fixed	

Description:

lastPremiumAccrualTime is initialized to block.timestamp when the wrapper is initialized. If the policy is bound later (e.g., days later), the first action that triggers _accruePremium() can incorrectly charge premium for the entire deployment -> binding interval , even though the policy wasn't active during that time.

In the "policy active" branch, _activeElapsedFrom(checkpoint) returns:

```function _activeElapsedFrom(uint256 checkpoint) internal view returns (uint256) { if (_isPolicyActive()) { return block.timestamp - checkpoint; }```

without clamping the checkpoint to the policy's startTime. So if checkpoint is earlier than startTime (or even earlier than binding time), the function counts that time as accrued premium time.

Example scenario

1.) Wrapper is initialized at Day 0 -> lastPremiumAccrualTime = Day 0.2.) Policy is only bound at Day 7 (no deposits were possible before this).3.) A user deposits after binding at Day 7.4.) _accruePremium() runs and computes elapsed time as Day 7 - Day 0 = 7 days, charging premium for 7 days, even though coverage didn't exist during that window.

This effectively makes users "pay premium retroactively" for time when the policy was not active.

Recommendations:

Clamp the effective checkpoint to startTime in the active policy branch

Customer Response:

Fixed in a2945e0.
```

Fix Review:

Fixed confirmed.

Low Severity Issues

L-01: lastKnownMorphoValue staleness creates an outdated _liabilityHeadroom value

Severity: Low	Impact: Low	Likelihood: Low
CoveredVaultAccrual.sol	Status: Acknowledged	

Description:

During _accruePremium() and _accruedPremiumFor(), the accrued amount is based on the underlying morpho value fetched via _morphoValue(). The _morphoValue() function has a fallback mechanism which resorts to the last known morpho value in case Morpho's `convertToAssets` function reverts.

```
function _morphoValue() internal view returns (uint256) {
        try
IVaultV2(morphoVault).convertToAssets(IVaultV2(morphoVault).balanceOf(address(thi
s))) returns (
            uint256 val
        ) {
            return val;
        } catch {
            return lastKnownMorphoValue;
        }
    }
```

The problem with this approach is that it will create an asymmetrical headroom value which may not reflect the real current value:

```
function _liabilityHeadroom(uint256 morphoVal) internal view returns (uint256) {
        return morphoVal > totalPremiumLiability ? morphoVal -
totalPremiumLiability : 0;
    }
```

Working with an outdated lastKnownMorphoValue may result in a smaller or greater headroom than it actually is.

Customer Response: Acknowledged.

L-02: Multiple instances of rounding directions in favor of the user

Severity: Low	Impact: Low	Likelihood: Low
Files: CoveredVaultWrapper.sol	Status: Fixed	

Description:

The rounding directions in _computeOwedAmounts and _calcOwedDebt (rounded down) are done in favor of the user, rather than in favor of the protocol.

_computeOwedAmounts calculates the amounts of premium that the user owes during withdrawals:

```
function _computeOwedAmounts(

        address owner_,

        uint256 shares_,

        uint256 ownerBalance_

    )
        private
        view
        returns (WithdrawalAmounts memory amounts)
    {

        uint256 principal = userPrincipal[owner_];
        amounts.totalOwedPremium = _calcOwedDebt(principal,
cumulativePremiumPerUnit, userPremiumDebt[owner_]);
        amounts.uncappedPremium = ownerBalance_ > 0
            ? amounts.totalOwedPremium.mulDiv(shares_, ownerBalance_,
Math.Rounding.Floor)
            : amounts.totalOwedPremium;
    }
```

Part of the above calculation is also _calcOwedDebt:

```
function _calcOwedDebt(uint256 principal, uint256 cumulative, uint256 userDebt)
private pure returns (uint256) {
        uint256 raw = principal.mulDiv(cumulative, WAD, Math.Rounding.Floor);
        return raw > userDebt ? raw - userDebt : 0;
    }
```

Another instance like this is `_capAndApplyDeductible` when calculating the deductible (rounding up would results in a greater one, i.e. it would be in favor of the protocol):

```
uint256 deductibleAmt = basis.mulDiv(deductibleBps, BPS_DENOMINATOR,
Math.Rounding.Floor);
```

Recommendations:

Round-up, instead of rounding down in the above-mentioned instances.

Customer Response:

Fixed in a2945e0.

Fix Review:

Fix confirmed.

L-03: totalAssets() can theoretically return 0 with outstanding shares, which would allow users to mint a disproportionate amount of shares

Severity: Low	Impact: Low	Likelihood: Low
Files: CoveredVaultWrapper.sol	Status: Acknowledged	

Description:

totalAssets can theoretically become 0 in cases when liabilities become great enough, while at the same time morphoValue has decreased significantly. In a scenario in which totalSupply is above 0, but totalAssets becomes 0, users could severely inflate shares, this is due to the way how totalAssets are calculated:

```
function totalAssets() public view override (ERC4626Upgradeable, IERC4626)
returns (uint256) {

        uint256 mv = _morphoValue();

        uint256 totalLiability = totalPremiumLiability + _accruedPremiumFor(mv);

        return mv > totalLiability ? mv - totalLiability : 0;

    }
```

To practically showcase the difference, in a theoretical situation in which totalSupply is 100_000e24 (due to the 6 decimal offset), and totalAssets is 0:

Depositing 100e18 assets:

```
function _convertToShares(uint256 assets, Math.Rounding rounding) internal view
virtual returns (uint256) {

        return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(),
totalAssets() + 1, rounding);

    }
```

Would result in the minting of 1e49 shares.

Recommendations:

Account for similar black swan events by either manually or programmatically switching off / pausing deposits in these cases.

Customer Response:

Acknowledged — theoretical edge case, low practical risk. By design.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.