PHOENIX
SMART SECURITY
BY TANU GUPTA

Open-Cover Insured Vaults Audit, Sherlock

Version 1.0

Tanu Gupta

January 24, 2026

Open Cover Insured Vaults, Sherlock

Tanu Gupta

Jan 24, 2026

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

- Table of Contents
- Protocol Summary
  - Core Architecture
  - Asynchronous Operations Model
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] Keeper Can Manipulate Redemption Amounts by Timing Settlements and Rate Increases

  - [H-2] Malicious Users Can DoS `settle()` by Front-Running with `requestDeposit()` and `cancelDepositRequest()`
- Medium
  - [M-1] Miners/Validators Can Manipulate `block.timestamp` to Reduce Premium Streaming
  - [M-2] Users Can Bypass `maxAssets >= assets` Validation by Front-Running Transactions That Reduce `totalAssets()`

Protocol Summary

The **CoveredMetavault** protocol is an insured vault system that wraps existing ERC-4626 yield vaults and adds insurance coverage on top. The protocol implements an asynchronous deposit and redemption model based on ERC-7540, extending OpenZeppelin's ERC-4626 implementation with premium streaming and batch settlement functionality.

Core Architecture

The protocol operates as a metavault layer that sits between users and underlying ERC-4626 vaults. Users deposit assets (e.g., bbqUSDC) into the `CoveredMetavault`, which then deposits those assets into a wrapped ERC-4626 yield vault. The protocol issues covered shares to users, representing their position in the insured vault.

Asynchronous Operations Model

Unlike standard ERC-4626 vaults where deposits and redemptions are instantaneous, `CoveredMetavault` uses a three-phase asynchronous model:

1. **Request Phase**: Users call `requestDeposit()` or `requestRedeem()` to submit asynchronous operations. Deposits aggregate into a single pending balance per controller, while redemptions create individual request IDs.
2. **Settlement Phase**: A keeper calls `settle()` to batch-process pending operations. During settlement:
   - Premium is streamed from the vault's assets based on time elapsed and the configured premium rate
   - Pending deposits are pre-minted as shares at the current exchange rate

   - Redemption requests are processed and assets are reserved for claim

3. **Claim Phase**: Users call `deposit()`/`mint()` or `redeem()`/`withdraw()` to claim their settled shares or assets.

Disclaimer

I, Tanu Gupta, make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of 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.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the Sherlock severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The audit was conducted on the following contracts:

- CoveredMetavault.sol
- Constants.sol
- PercentageLib.sol
- interfaces/ICoveredMetavault.sol
- interfaces/IERC7540.sol

- interfaces/IERC7540Cancel.sol
- interfaces/IERC7575.sol

Roles

The protocol implements strict role separation:

- **Owner**: Upgrade implementation, grant/revoke roles
- **Manager**: Configure premium collector, set premium rate (<= max), set minimum request assets
- **Keeper**: Execute `settle()` operations, batch-settle redemptions, push claimable shares/assets, cancel deposits
- **Guardian**: Pause/unpause operations for emergency response

Executive Summary

The audit is conducted on the OpenCover repository. The audit duration was from Jan 19, 2026 to Jan 23, 2026.

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	0
Info	0
Total	4

Findings

High

[H-1] Keeper Can Manipulate Redemption Amounts by Timing Settlements and Rate Increases

Description The keeper controls when `settle()` is called to process redemption requests. In `settle()`, premium is streamed FIRST, then redemptions are settled at the lower asset amount.

By strategically timing settlements, the keeper can systematically extract value from users:

1. **Delayed Settlement:** The keeper can delay settlement for 24 hours to let premium accumulate, then settle redemptions after premium has been streamed, making users redeem fewer assets.
2. **Strategic Timing After Rate Increases:** If `setPremiumRateBps()` was just called (increasing the rate), the keeper can immediately call `settle()` to stream premium at the NEW higher rate, causing a significant double premium hit (old rate + new rate) before redemptions are settled, causing a significant loss to users This is a trusted party abuse issue where the keeper can systematically extract value from users by controlling settlement timing.

Root Cause

The **settlement order and keeper timing control allows the keeper to systematically extract value from users**:

```
function settle(uint256 expectedPendingAssets, uint256[] calldata redeemRequestIds) external {
    // ...
    _streamPremium();  // Premium streams FIRST (line 771)

    // ... epoch advancement ...

    // Settle redemption requests
    for (uint256 i = 0; i < redeemRequestIdsLength; ++i) {
        // ...
        assets = _convertToAssets(shares, Math.Rounding.Floor);  // Uses LOWER totalAssets() after premium (line 819)
        // ...
    }
}
```

1. `_streamPremium()` is responsible for extracting premium, causing totalAssets() to reduce.
2. `uint256 assets = _convertToAssets(shares, Math.Rounding.Floor);` - Asset amounts are locked using `_convertToAssets()` which uses the LOWER

`totalAssets()` after premium has been streamed
3. If `setPremiumRateBps()` was just called (increasing the rate), keeper can immediately call `settle()` to stream premium at the NEW higher rate, causing a significant double hit to `totalAssets()` before settling redemptions

The keeper can systematically make users redeem fewer assets by:

1. Delaying settlement to accumulate premium
2. Timing settlement right after rate increases to maximize premium extraction

Internal Pre-conditions

1. Users must have called `requestRedeem()` with shares
2. Redemption requests must not be settled yet
3. Premium rate must be non-zero - `premiumRateBps > 0`
4. `block.timestamp > lastPremiumTimestamp` - time must have elapsed since last premium stream
5. Keeper must be authorized to call `settle()`
6. Contract must be active:
    - Not paused
    - Initialized

External Pre-conditions

1. Keeper must be willing to extract value from users

Attack Path

1. **Delayed Settlement**

- Calls `requestRedeem(shares)`, shares locked in vault
- Keeper waits to let premium accumulate (e.g., 1 day)
- Keeper calls `settle(..., [requestId])` - Premium streams FIRST, reducing `totalAssets()`
- User gets fewer assets systematically less than if settled immediately

2. Strategic Timing After Rate Increase

- Manager calls `setPremiumRateBps(newRate)` - Streams premium at OLD rate, updates to NEW (higher) rate
- Keeper immediately calls `settle()` - Premium streams AGAIN at the NEW (higher) rate
- `totalAssets()` decreases significantly (premium at old rate + premium at new rate)
- User gets significantly fewer assets due to amounts locked at much lower `totalAssets()`

Impact

1. The keeper (trusted party) can systematically extract value from users by controlling settlement timing. This is a direct financial extraction from users.
2. Affects EVERY redemption settlement where keeper controls timing.
3. Keeper can extract (0.1-1%+) per redemption by delaying settlement
4. Keeper can extract significantly more (0.2-2%+) by timing settlement after rate increases (double premium streaming)
5. For large redemptions, the impact is significant
6. **Violates trust assumption**: The keeper is supposed to act in users' best interests, but can abuse their position

Proof of Concepts

Paste the following test cases in Redemption.t.sol to simulate the finding:

```
/// @dev This test shows Scenario 1 : Delayed Settlement
/// @dev Compares 0.5 day delay vs 1 day delay to show timing impact
function test_KeeperManipulation_DelaysSettlementToExtractValue() public {
        // Setup: Set premium rate and initialize premium timestamp
        uint256 initalTimestamp = block.timestamp;
        _setPremiumRate(500); // 5% annual

        // Mint shares for owner and other
        uint256 shares1_Owner = _mintSharesTo(owner, DEPOSIT_AMOUNT * 10);
        uint256 shares1_Other = _mintSharesTo(other, DEPOSIT_AMOUNT * 10);

        // Owner and other request redemption at the same time
        vm.prank(owner);
        uint256 reqId1_Owner = vault.requestRedeem(shares1_Owner, owner, owner);
        vm.prank(other);
        uint256 reqId1_Other = vault.requestRedeem(shares1_Other, other, other);

        // Advance time by 0.5 days
        // Settle redemption for owner
        vm.warp(initalTimestamp + 0.5 days);
        _settle(0, reqId1_Owner);

        // Get claimable assets after settlement for owner
        uint256 claimableAfter_Owner = vault.maxWithdraw(owner);

        // Advance time by 1 days
        vm.warp(initalTimestamp + 1 days);
        _settle(0, reqId1_Other);
```

```
        // Get claimable assets after settlement for other
        uint256 claimableAfter_Other = vault.maxWithdraw(other);

        // Verify owner should get more claimable assets than other
        assertGt(claimableAfter_Owner, claimableAfter_Other, "Owner should get more claimable assets than other");
    }

/// @dev This test shows Scenario 2: Strategic Timing After Rate Increase
///@dev Demonstrates how keeper can extract more value by timing settlement after rate increase.
 function test_KeeperManipulation_TimesSettlementAfterRateIncrease() public {
        uint256 initalTimestamp = block.timestamp;
        // Setup: Set initial premium rate and create settled deposits
        _setPremiumRate(500); // 5% annual
        uint256 shares = _mintSharesTo(owner, DEPOSIT_AMOUNT * 10); // 10,000 assets
        uint256 initialTotalAssets = vault.totalAssets();

        // User requests redemption
        vm.prank(owner);
        uint256 reqId = vault.requestRedeem(shares, owner, owner);

        // Calculate what user would get if settled before rate increase
        uint256 totalAssetsBeforeRateIncrease = vault.totalAssets();
        uint256 sharesToRedeem = shares;
        uint256 assetsIfSettledBeforeRateIncrease =
            Math.mulDiv(sharesToRedeem, totalAssetsBeforeRateIncrease, vault.totalSupply(), Math.Rounding.Floor);

        vm.warp(initalTimestamp + 0.5 days);
        // Manager increases premium rate (doubles it)
        // This streams premium at OLD rate (5%), then updates to NEW rate (10%)
        uint256 collectorBalanceBeforeRateIncrease = asset.balanceOf(premiumCollector);
        _setPremiumRate(1000); // 10% annual (doubled)
        uint256 collectorBalanceAfterRateIncrease = asset.balanceOf(premiumCollector);
        uint256 premiumFromRateChange = collectorBalanceAfterRateIncrease - collectorBalanceBeforeRateIncrease;

        // Keeper immediately calls settle() after rate increase
        // This streams premium AGAIN at NEW rate (10%), causing double premium hit
        uint256 collectorBalanceBeforeSettle = asset.balanceOf(premiumCollector);
```

```
        vm.warp(initalTimestamp + 0.5 days + 10 seconds);
        _settle(0, reqId);
        uint256 collectorBalanceAfterSettle = asset.balanceOf(premiumCollector);

        // Calculate what user actually gets
        uint256 assetsUserGets = vault.maxWithdraw(owner);

        // Calculate total premium streamed (from rate change + from settle)
        uint256 premiumFromSettle = collectorBalanceAfterSettle - collectorBalanceBeforeSettle;
        uint256 totalPremiumStreamed = premiumFromRateChange + premiumFromSettle;

        // User gets significantly less than if settled before rate increase
        assertLt(
            assetsUserGets,
            assetsIfSettledBeforeRateIncrease,
            "User should get less assets after rate increase and delayed settlement"
        );

        // Calculate extraction amount
        uint256 extraction = assetsIfSettledBeforeRateIncrease - assetsUserGets;

        // Log the results for visibility
        console.log("=== Keeper Timing Manipulation: After Rate Increase ===");
        console.log("Initial totalAssets:", initialTotalAssets);
        console.log("Shares to redeem:", sharesToRedeem);
        console.log("Assets if settled before rate increase:", assetsIfSettledBeforeRateIncrease);
        console.log("Assets user actually gets:", assetsUserGets);
        console.log("Premium from rate change (5%):", premiumFromRateChange);
        console.log("Premium from settle (10%):", premiumFromSettle);
        console.log("Total premium streamed:", totalPremiumStreamed);
        console.log("Value extracted from user:", extraction);

        // Verify extraction is significant (more than single premium stream)
        assertGt(extraction, 0, "Keeper should extract value by timing after rate increase");
        assertGt(totalPremiumStreamed, premiumFromRateChange, "Total premium should be more than just rate change");
        assertGt(extraction, premiumFromRateChange, "Extraction should be more than single premium stream");
    }
```

Recommended mitigation

1. Enforce a maximum delay between redemption request and settlement:

```
function settle(...) external {
    // ... existing code ...

    // Check maximum delay for redemptions
    for (uint256 i = 0; i < redeemRequestIdsLength; ++i) {
        RedeemRequestStorage storage redeemRequestStorage = $.
            redeemRequests[redeemRequestIds[i]];
        uint256 requestAge = block.timestamp - redeemRequestStorage.
            timestamp;
        require(requestAge <= MAX_SETTLEMENT_DELAY, "Settlement delay
            exceeded");

        // ... settle redemption ...
    }

    // ... rest of function ...
}
```

2. Disincentive the Keeper for deliberately delaying the `settle()` call

[H-2] Malicious Users Can DoS `settle()` by Front-Running with `requestDeposit()` and `cancelDepositRequest()`

Description

The `settle()` function uses `expectedPendingAssets` to validate that `totalPendingAssets` matches the expected value, reverting if they don't match at L-729

A malicious user can front-run the keeper's `settle()` call by manipulating `totalPendingAssets` through `requestDeposit()` and `cancelDepositRequest()`, causing the settlement to revert.

By repeatedly front-running with alternating deposit requests and cancellations, the attacker can prevent `settle()` from ever succeeding, causing a complete DoS of the settlement mechanism.

This blocks new epochs, prevents deposits from becoming claimable, and prevents redemptions from being settled, effectively halting core protocol operations

Root Cause

The `settle()` function validates that `totalPendingAssets` matches the expected value:

```
function settle(uint256 expectedPendingAssets, uint256[] calldata
    redeemRequestIds)
```

```
    external override whenNotPaused nonReentrant onlyRole(KEEPER_ROLE)
{
    VaultStorage storage $ = _getVaultStorage();

    uint256 pendingAssetsTotal = $.totalPendingAssets;
    if (expectedPendingAssets != 0) {
        require(
            pendingAssetsTotal == expectedPendingAssets,  // Can be
                manipulated by front-running
            UnexpectedPendingAssets(expectedPendingAssets,
                pendingAssetsTotal)
        );
    }

    // ... rest of settlement logic
}
```

The root cause is the **reliance on `expectedPendingAssets` validation combined with public state-modifying functions that can be front-run**. The protocol attempts to prevent manipulation by requiring the keeper to specify expected pending assets, but this creates a vulnerability where:

- The validation is too strict
- Public functions can manipulate the state being validated
- Front-running is trivial with MEV capabilities
- No protection against repeated manipulation

Internal Pre-conditions

1. Keeper must call settle() with non-zero `expectedPendingAssets`-`expectedPendingAssets != 0`
2. Attacker must have assets to deposit
3. Deposit must be in current epoch to cancel
    - `depositStorage.pendingAssets > 0`
    - `depositStorage.lastSyncedEpoch == currentEpoch`
4. Contract must be active:
    - Not paused
    - Initialized

External Pre-conditions

1. `settle()` transaction must be visible in mempool
2. Attacker must have ability to submit transactions with higher gas price
3. Attacker must be willing to pay gas costs

Attack Path

1. Keeper calls `settle(expectedPendingAssets = 1000, ...)` - transaction visible in mempool
2. Attacker front-runs by calling `requestDeposit(500)` - `totalPendingAssets` increases from 1000 to 1500
3. `settle()` executes and reverts
4. Keeper tries again with `settle(expectedPendingAssets = 1500, ...)`
5. Attacker front-runs again by calling `cancelDepositRequest()` - `totalPendingAssets` decreases from 1500 to 1000
6. `settle()` executes and reverts again
7. Attacker can keep alternating to prevent any successful settlement

Impact

1. The validation was explicitly added to "prevent manipulation". Bypassing it defeats this purpose.
2. If the keeper wants to use the validation feature (which is the intended design, passing `expectedPendingAssets != 0`), the DoS is possible.
3. While the keeper can work around the DoS by setting `expectedPendingAssets`, this forces them to remove an intended safety feature. The vulnerability should still be fixed properly rather than relying on a workaround that defeats the purpose of the validation.
4. Complete Blocking of Settlement Operations (when validation is used `expectedPendingAssets != 0`) and hence the complete protocol.

Proof of Concepts

Paste the following code in Deposit.t.sol to check out the vulnerability:

```
function test_User_DOS_Settle_By_Requesting_Deposit_And_Cancelling_It() public {
        vm.prank(owner);
        vault.requestDeposit(DEPOSIT_AMOUNT, owner, owner);

        uint256 totalPendingAssetsBefore = vault.totalPendingAssets();
        assertEq(totalPendingAssetsBefore, DEPOSIT_AMOUNT);

        // User front runs the keeper's settle call by requesting a deposit
        vm.prank(other);
        vault.requestDeposit(DEPOSIT_AMOUNT, other, other);

        vm.prank(keeper);
        // Keeper should revert because the total pending assets is not equal to the expected pending assets
        vm.expectRevert(abi.encodeWithSelector(ICoveredMetavault.UnexpectedPendingAssets.selector, DEPOSIT_AMOUNT,
```

```
            totalPendingAssetsBefore+DEPOSIT_AMOUNT));
        vault.settle(DEPOSIT_AMOUNT, new uint256[](0));

        // User cancels the deposit request, front runs the keeper's settle call by requesting a cancel deposit
        vm.prank(other);
        vault.cancelDepositRequest(DEPOSIT_REQUEST_ID, other, other);

        vm.prank(keeper);
        // Keeper should revert because the total pending assets is not equal to the expected pending assets
        vm.expectRevert(abi.encodeWithSelector(ICoveredMetavault.UnexpectedPendingAssets.selector, DEPOSIT_AMOUNT+DEPOSIT_AMOUNT, DEPOSIT_AMOUNT));
        vault.settle(DEPOSIT_AMOUNT+DEPOSIT_AMOUNT, new uint256[](0));

    }
```

Recommended mitigation

1. Remove `expectedPendingAssets` Validation - If the validation is not critical, remove it entirely:

```
function settle(uint256 expectedPendingAssets, uint256[] calldata redeemRequestIds)
    external override whenNotPaused nonReentrant onlyRole(KEEPER_ROLE)
{
    VaultStorage storage $ = _getVaultStorage();

    // Remove the validation check
    // uint256 pendingAssetsTotal = $.totalPendingAssets;
    // if (expectedPendingAssets != 0) { ... }

    uint256 pendingAssetsTotal = $.totalPendingAssets;

    // ... rest of settlement logic
}
```

2. Snapshot totalPendingAssets at Transaction Start - Use a snapshot mechanism to prevent manipulation:

```
function settle(uint256 expectedPendingAssets, uint256[] calldata redeemRequestIds)
    external override whenNotPaused nonReentrant onlyRole(KEEPER_ROLE)
{
    VaultStorage storage $ = _getVaultStorage();

    // Snapshot at start of transaction (before any state changes)
    uint256 pendingAssetsTotal = $.totalPendingAssets;

```

```
    // Lock deposits for this settlement (prevent new deposits/
        cancellations)
    $.settlementInProgress = true;

    if (expectedPendingAssets != 0) {
        require(
            pendingAssetsTotal == expectedPendingAssets,
            UnexpectedPendingAssets(expectedPendingAssets,
                pendingAssetsTotal)
        );
    }

    // ... rest of settlement logic

    $.settlementInProgress = false;
}

// Add check to requestDeposit and cancelDepositRequest
function requestDeposit(...) external override {
    VaultStorage storage $ = _getVaultStorage();
    require(!$.settlementInProgress, "Settlement in progress");
    // ... rest of function
}
```

Medium

[M-1] Miners/Validators Can Manipulate `block.timestamp` to Reduce Premium Streaming

Description

The `CoveredMetavault.sol::_streamPremium()` function in uses `block.timestamp` directly to calculate the duration for premium streaming.

Miners/validators can manipulate `block.timestamp` within protocol bounds (typically +-15 seconds on Ethereum, varies on other chains) to reduce the calculated duration, resulting in less premium being streamed to the protocol.

Root Cause

The premium streaming calculation relies on `block.timestamp` without any validation or bounds checking:

```
function _streamPremium() internal returns (uint256 assetsStreamed,
    uint64 duration) {
    VaultStorage storage $ = _getVaultStorage();
```

```
    uint64 lastPremiumTimestamp = $.lastPremiumTimestamp;
    uint64 nowTimestamp = uint64(block.timestamp);  // Direct use of
        manipulatable timestamp

    if (lastPremiumTimestamp != 0 && nowTimestamp >
        lastPremiumTimestamp) {
        duration = nowTimestamp - lastPremiumTimestamp;  //
            Manipulatable duration

        // ... premium calculation based on duration ...
        uint256 premium = assetsAfter.annualBpsProRata(annualRateBps,
            remainingSeconds);
    }
}
```

The root cause of this vulnerability is the direct and unvalidated use of block.timestamp for time-sensitive financial calculations.

The protocol assumes block.timestamp accurately represents the passage of time, but this assumption is incorrect because:

- Miners/validators have control over `block.timestamp` within protocol-defined bounds. On Ethereum, validators can set timestamps within +-15 seconds of the previous block's timestamp, and in practice, this window can be larger.
- The premium calculation formula directly uses the duration derived from timestamps:

```
duration = nowTimestamp - lastPremiumTimestamp;
premium = assetsAfter.annualBpsProRata(annualRateBps, remainingSeconds)
    ;
```

Any manipulation of duration directly affects the premium amount.

Internal Pre-conditions

1. Vault must have settled assets - `totalAssets()` must return a non-zero value
2. Premium rate must be non-zero - `premiumRateBps` must be greater than 0
3. Previous premium timestamp must exist - `lastPremiumTimestamp` must be non-zero
4. Time must have advanced - `block.timestamp > lastPremiumTimestamp`
5. `settle()` or `setPremiumRateBps` function must be callable for calling `_streamPremium`
6. Keeper must have `KEEPER_ROLE`
7. Vault must be in `active` state
8. Contract must be `initialized`
9. Contract must not be `paused`

External Pre-conditions

1. Attacker must be a validator/miner with the ability to propose blocks
2. Attacker must be able to influence a validator/miner
3. `settle()` or `setPremiumRateBps` transaction must be visible in mempool before inclusion
4. Timestamp can be set within (+-) 15 seconds of previous block

Attack Path

1. Validator sees `settle()` transaction in mempool - The keeper calls `settle()` which triggers `_streamPremium()`
2. Validator manipulates `block.timestamp` - Sets timestamp to be earlier (typically -15 seconds)
3. Transaction included with manipulated timestamp - Duration calculation is reduced
4. Protocol receives less revenue than expected
5. Validator can repeat this across multiple settlements

Impact

1. Protocol receives less premium than intended, affecting insurance coverage funding
2. Small losses compound over time with frequent settlements

Proof of Concepts

Paste the following code in Deposits.t.sol to see the results:

```
 /// @dev Shows that timestamp manipulation within protocol bounds
     reduces premium collected
function test_Miner_Maniputale_timestamp_to_reduce_premium_streaming()
     public {
        _setPremiumRate(500); // 5% annual

        // Setup: Create deposits and settle to initialize premium
            timestamp
        vm.prank(owner);
        vault.requestDeposit(DEPOSIT_AMOUNT, owner, owner);
        vm.prank(other);
        vault.requestDeposit(DEPOSIT_AMOUNT, other, other);

        // Initial settlement (sets lastPremiumTimestamp, but streams 0
             premium on first epoch)
        vm.prank(keeper);
        vault.settle(DEPOSIT_AMOUNT + DEPOSIT_AMOUNT, new uint256[](0))
            ;

        // Store the timestamp after initial settlement (this becomes
            lastPremiumTimestamp)
        uint256 lastPremiumTimestamp = block.timestamp;

        // Advance time by 1 day normally
        uint256 oneDay = uint256(1 days);
```

```
        uint256 normalTimestamp = lastPremiumTimestamp + oneDay;
        uint256 manipulatedDuration = uint256(normalTimestamp - 15 seconds);
        console.log("manipulatedDuration: ", manipulatedDuration, normalTimestamp);
        vm.warp(normalTimestamp);

        // Capture assets BEFORE settlement (this is what premium will be calculated on)
        uint256 assetsBeforeSettlement = vault.totalAssets();

        // Scenario 1: Normal settlement (no manipulation) - 1 day = 86400 seconds
        vm.prank(keeper);
        vault.settle(0, new uint256[](0));

        // Calculate expected premium with normal duration (1 day = 86400 seconds)
        uint256 expectedPremiumNormal = PercentageLib.annualBpsProRata(assetsBeforeSettlement, 500, uint64(oneDay));
        // Scenario 2: Miner manipulates timestamp by -15 seconds (within protocol bounds)
        // This reduces the duration calculation from 86400 to 86385 seconds
        // this is the duration that the miner will manipulate the timestamp to
        // normal timestamp - 15 seconds
        uint256 manipulatedDuration_seconds = 86385 seconds;

        // Calculate expected premium with manipulated duration
        uint256 expectedPremiumManipulated = PercentageLib.annualBpsProRata(assetsBeforeSettlement, 500, uint64(manipulatedDuration_seconds));
        uint256 loss = expectedPremiumNormal - expectedPremiumManipulated;

        assertGt(loss, 1, "Loss should be measurable");
    }
```

Recommended mitigation

1. Implement Minimum Duration Checks - Add bounds checking to prevent excessive manipulation:

```
function _streamPremium() internal returns (uint256 assetsStreamed, uint64 duration) {
    VaultStorage storage $ = _getVaultStorage();
    uint64 lastPremiumTimestamp = $.lastPremiumTimestamp;
    uint64 nowTimestamp = uint64(block.timestamp);

    if (lastPremiumTimestamp != 0 && nowTimestamp > lastPremiumTimestamp) {
```

```
        duration = nowTimestamp - lastPremiumTimestamp;

        // Mitigation: Enforce minimum duration to prevent excessive
           manipulation
        uint64 MIN_SETTLEMENT_DURATION  = ... ; // appropriate minimum
        require(
            duration >= MIN_SETTLEMENT_DURATION,
            "Duration too short, possible timestamp manipulation"
        );

        // ... rest of premium calculation ...
    }
}
```

2. Use Oracle-Based Time (For Critical Calculations) - For high-value vaults, consider using Chainlink or similar oracle for time:

```
import {AggregatorV3Interface} from "@chainlink/contracts/src/v0.8/
   interfaces/AggregatorV3Interface.sol";

function _streamPremium() internal returns (uint256 assetsStreamed,
   uint64 duration) {
    // Use oracle timestamp for critical calculations
    uint64 nowTimestamp = _getOracleTimestamp(); // From Chainlink or
       similar

    // ... rest of calculation ...
}
```

[M-2] Users Can Bypass `maxAssets >= assets` Validation by Front-Running Transactions That Reduce `totalAssets()`

Description

The `_claimDeposit()` function validates that `maxAssets >= assets`, where maxAssets is calculated using the current `totalAssets()` value.

However, `claimableShares` is fixed from the epoch snapshot at settlement time. This creates a validation mismatch where users can bypass the check by front-running any transaction that reduces `totalAssets()` (such as `setPremiumRateBps()` or `settle()` calling `_streamPremium()`).

Core Issue: The validation uses dynamic `totalAssets()` but `claimableShares` is fixed, creating a bypass opportunity.

Root Cause

The `CoveredMetaVault::_claimDeposit()` function has a validation mismatch:

```
function _claimDeposit(uint256 assets, address receiver, address controller) internal returns (uint256 shares) {
    // ...
    uint256 claimableShares = depositStorage.claimableShares;  // Fixed from epoch snapshot
    require(claimableShares != 0, InsufficientClaimableAssets(controller, 0, assets));

    // Maximum assets this controller can take right now.
@>    uint256 maxAssets = _convertToAssets(claimableShares, Math.Rounding.Floor);  // Uses CURRENT totalAssets()
@>    require(maxAssets >= assets, InsufficientClaimableAssets(controller, maxAssets, assets));  // Validation can be bypassed

    // Convert requested assets to shares at current price.
    shares = _convertToShares(assets, Math.Rounding.Floor);
    require(claimableShares >= shares, InsufficientClaimableShares(controller, claimableShares, shares));
    // ...
}
```

- `maxAssets = _convertToAssets(claimableShares)` uses the current `totalAssets()` value.
- If `totalAssets()` decreases (e.g., from premium streaming), maxAssets decreases, potentially causing `deposit()` to revert if `maxAssets < requestedAssets`.
- Front-running transactions that reduce `totalAssets()` allows users to claim before the validation check fails.

The **validation mismatch between fixed `claimableShares` and dynamic `maxAssets` calculation** creates a bypass opportunity.

Operations that reduce `totalAssets()` include:

1. `setPremiumRateBps()` L-918 - Calls `_streamPremium()` which reduces `totalAssets()`
2. `settle()` L-838 - Calls `_streamPremium()` which reduces `totalAssets()`

Internal Pre-conditions

1. Transaction that reduces `totalAssets()` must be pending
    - Manager calls `setPremiumRateBps()`
    - OR keeper calls `settle()` (calls `_streamPremium()`)
2. User must have claimable deposits - `claimableShares > 0`

3. Premium rate must be non-zero - `premiumRateBps > 0`
4. Time must have elapsed since last premium stream - `block.timestamp > lastPremiumTimestamp`
5. Contract must be active
   - Not paused
   - Initialized

External Pre-conditions

1. Transactions that reduce `totalAssets()` (e.g., `setPremiumRateBps()`, `settle()`) must be visible in mempool
2. User must have ability to submit transactions with higher gas price
3. Benefit from front-running must exceed gas costs

Attack Path

1. User has claimable deposits - settlement already happened, `claimableShares` is fixed from epoch snapshot
2. Transaction that reduces `totalAssets()` is pending - `setPremiumRateBps()` or `settle()`
3. User front-runs by calling `deposit()` - claims before `totalAssets()` is reduced
4. Transaction executes - `totalAssets()` decreases, maxAssets decreases
5. User bypassed validation else transaction would have failed after `totalAssets()` reduction (if `maxAssets < requestedAssets`)

Impact

1. Front-runner doesn't get MORE shares (they're fixed from epoch snapshot), but avoids reversion
2. MEV-capable users avoid transaction failures, while others must retry with adjusted amounts
3. Users have a way out of the validation, making the validation ineffective.

Proof of Concepts

Paste the following code in Deposits.t.sol to run the test case:

```
///@dev Users can bypass `maxAssets >= assets` validation by front-running transactions that reduce `totalAssets()`
    function test_FrontRunTotalAssetsReduction_BypassesValidation() public {
        // Setup: User has claimable deposits from previous settlement
        uint256 depositAmount = 1000e18;
        _requestDeposit(depositAmount, owner, owner);
        _settle();

```

```
        // Get claimable shares (fixed from epoch snapshot)
        uint256 claimableShares = vault.claimableDepositRequest(
            DEPOSIT_REQUEST_ID, owner);
        assertGt(claimableShares, 0, "User should have claimable shares
            ");

        // Calculate maxAssets before totalAssets() reduction
        uint256 maxAssetsBefore = vault.maxDeposit(owner);

        // Advance time to allow premium streaming
        uint256 newTimestamp = block.timestamp + 1 days;
        vm.warp(newTimestamp);

        // Scenario 1: Front-runner claims BEFORE totalAssets() is
            reduced
        // Front-runner successfully bypasses validation
        _deposit(maxAssetsBefore, owner, owner);


        // Setup fresh scenario for "user who waits"
        _requestDeposit(depositAmount, other, other);
        _settle();
        vm.warp(newTimestamp + 1 days);
        //Streams premium to reduce totalAssets() further
        _setPremiumRate(1000);
        // User tries to claim the same amount - should revert
        vm.expectRevert();
        vm.prank(other);
        vault.deposit(maxAssetsBefore, other, other);

        console.log("Result: REVERTS - Validation fails (maxAssets <
            requestedAssets)");
    }
```

Recommended mitigation

Use Epoch Snapshot Price for Validation: Calculate `maxAssets` using the epoch snapshot price instead of current `totalAssets()`:

```
function _claimDeposit(uint256 assets, address receiver, address
    controller) internal returns (uint256 shares) {
    // ...
    _syncEpoch(controller);

    DepositStorage storage depositStorage = _getVaultStorage().deposits
        [controller];
    uint256 claimableShares = depositStorage.claimableShares;
    require(claimableShares != 0, InsufficientClaimableAssets(
        controller, 0, assets));

    // Use epoch snapshot price for maxAssets (matching fixed
```

```
        claimableShares)
    uint64 lastSyncedEpoch = depositStorage.lastSyncedEpoch;
    EpochAllocation storage epochStorage = $.epochAllocations[
        lastSyncedEpoch];
    uint256 epochTotalAssets = epochStorage.totalAssets;
    uint256 epochTotalShares = epochStorage.totalShares;

    // Calculate maxAssets using epoch snapshot price
    uint256 maxAssets = claimableShares.mulDiv(epochTotalAssets,
        epochTotalShares, Math.Rounding.Floor);
    require(maxAssets >= assets, InsufficientClaimableAssets(controller
        , maxAssets, assets));

    // ... rest of function
}
```