Origami Finance Morpho Auto-Compounder Security Review

By Jacopod Audits

X

Auditors

Lead Security Researcher: Jacobo Lansac (@Jacopod)

July 22, 2025

Contents

1 Introduction

A time-boxed review of **Morpho Auto-Compounder** smart contracts, developed by Origami Finance. The focus of this review is to identify security vulnerabilities, explain their root-cause and provide solutions to mitigate the risk.

Gas optimizations are not the main focus but will also be identified if found.

2 About Jacopod

Jacopod Audits provides thorough security assessments for smart contract applications, combining deep manual review with advanced testing techniques like fuzzing and invariant testing to identify and mitigate potential vulnerabilities before deployment across diverse DeFi protocols. Here are some highlights:

- Ranked #6 all-time in Hats Finance competitions (Leaderboard) by number of issues found.
- Secured protocols with more than $200M TVL in total:
  - Origami_fi, $111M TVL.
  - Templedao, $41M TVL.
  - PinLinkAi, $21M TVL.
  - Vectorreserve, $45M peak TVL (now inactive).
- Check the Complete Audit Portfolio.

Contact for audits:

- Telegram: @jacopod_eth
- X: @jacopod

3 About Origami Finance

Origami Finance is a novel tokenised leverage protocol on Ethereum and Berachain. Fully integrated with third-party lenders, Origami allows users to achieve non-custodial portfolio exposure to popular looping and other yield strategies via a familiar vault UX. Origami vaults are fully automated to eliminate tedious reward harvesting, maximise capital efficiency, and minimise liquidation risk for the underlying position.

4 Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time and resource-bound effort to find as many vulnerabilities as possible, but there is no guarantee that all issues will be found. This security review does not guarantee against a hack. Any modifications to the code will require a new security review.

This review does not focus on the correctness of the happy paths. Instead, it aims to identify potential security vulnerabilities and attack vectors derived from an unexpected and harmful usage of the contracts. The devs are ultimately responsible for the correctness of the code and its intended functionality.

A security review is not an endorsement of the underlying business or product and can never be taken as a guarantee that the protocol is bug-free.

5 Risk classification

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

5.1 Impact

- **High**: leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium**: only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low**: can lead to unexpected behavior with some of the protocol's functionalities that are not so critical.

5.2 Likelihood

- **High**: almost certain to happen, easy to perform or highly incentivized.
- **Medium**: only conditionally possible or incentivized, but still relatively likely
- **Low**: requires multiple unlikely conditions, or little-to-no incentive

5.3 Action required for severity levels

- **Critical**: Must fix as soon as possible (if already deployed)
- **High**: Must fix (before deployment if not already deployed)
- **Medium**: Should fix
- **Low**: Could fix

6 Executive Summary

Summary

Project Name	Origami Finance
Type of Project	Auto Compounding ERC4626 Vault
Repository	TempleDAO/origami/
Review commit	267f0c956079ca2c4a27f9745bc817bd80d75416
Mitigation commit	3ff9f5a5e984cf2ebeb30b387aa9b0119744ff90
Audit Timeline	2025-07-15 - 2025-07-22
Methods	Manual Review, Testing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	2

6.1 Files in scope

Files in scope	Lines Changed	Notes
OrigamiErc4626WithRewardsManager.sol	408 additions	Major change
OrigamiVestingReserves.sol	111 additions	Major change
OrigamiErc4626.sol	34 lines	Smol Updates
OrigamiDelegated4626Vault.sol	71 lines	Smol Updates
OrigamiInfraredVaultManager.sol	84 lines	Smol Updates
OrigamiBoycoVault.sol	69 deletions	Smol Updates
IOrigamiErc4626WithRewardsManager.sol	91 additions	Interface addition
IMerklDistributor.sol	28 additions	Interface addition
IOrigamiVestingReserves.sol	30 additions	Interface addition

7 Architecture review

7.1 Architecture Overview

The notes below relate to the specific Morpho Auto-Compounder vault.

Vault (OrigamiDelegated4626Vault):

- An ERC4626-compliant vault accepting USDS deposits
- The vault issues shares to deposits but delegates the assets to a manager contract

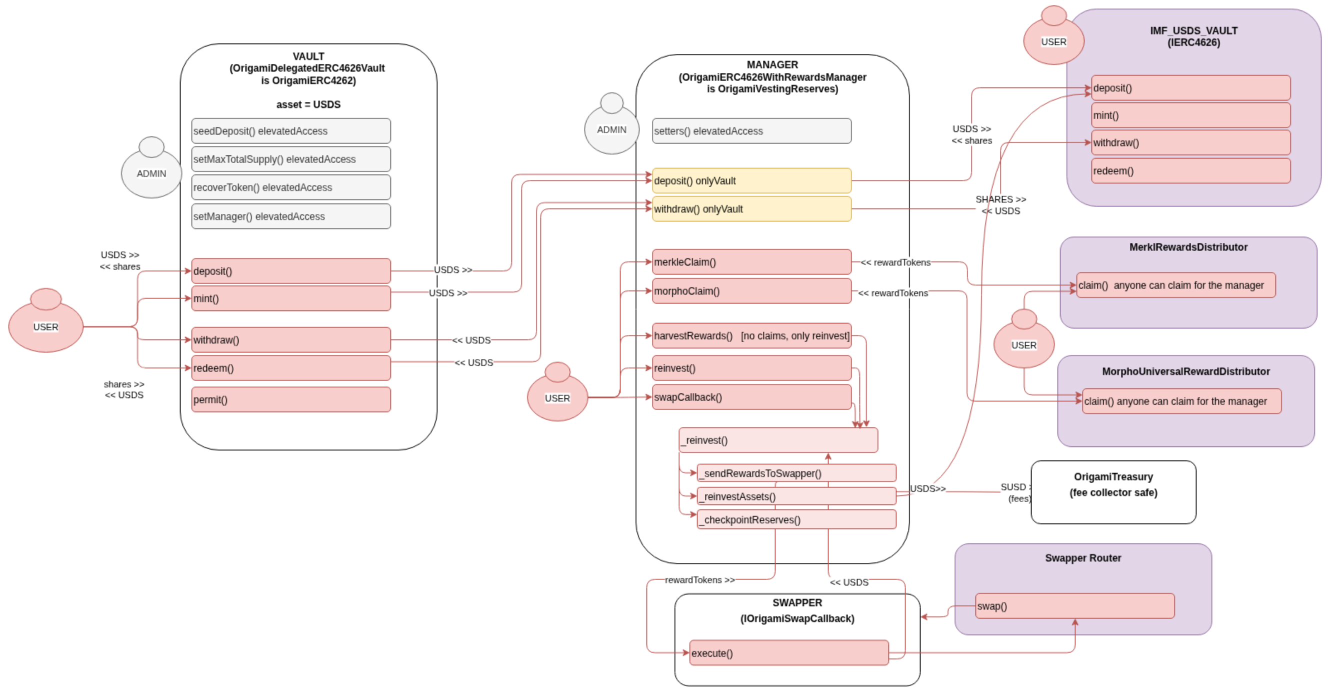
Manager (OrigamiErc4626WithRewardsManager):

- Manages vault assets by depositing into the underlying Morpho IMF-USDS vault (another ERC4626 vault).
- Handles reward distribution from Merkl and Morpho distributors.
- Rewards come in different tokens, which are sent to a swapper, and then received as assets
- These assets are then reinvested in the underlying vault, applying a performance fee first.
- Reinvested assets have a vesting mechanism (possibly configured to 1 day) to avoid step-wise jumps in the Origami vault share price
- Hardcoded depositFee to 0%, and configurable withdrawal fee.

7.1.1 Architecture assessment

The architecture separates user-facing vault operations from the reward harvesting and fee management logic.

- The Vault delegates all asset management to manager via setManager(), and can move the funds to a different manager.
- Manager deposits user funds into underlying IMF_USDS_VAULT
- Rewards flow through distributors → manager → swapper → manager → vault
- Performance fees are collected right before reinvesting into the underlying vault



Architecture diagram of Morpho Auto-Compounder

7.1.2 Centralization

Origami admins have the power to update the manager of the vault via 'setManager()'. This function can withdraw all user funds from a manager and move to another one. This means that the admin has the power to move all user deposits to a fake vault contract and effectively steal all user funds.

Although Origami has a strong reputation and the scenario in which they steal funds is unlikely, compromised private keys of Origami’s multisig could put user deposits at risk.

8 Findings

8.1 Medium Risk

8.1.1 [M1] - Wrong calculation of `totalAssets` including `unallocatedAssets` can cause a share price drop when calling `deposit()` when there are unallocated assets

The `manager.totalAssets()` is used by the `ERC4626.totalAssets()` to calculate the share price. Here is how the `totalAssets()` is calculated by the `OrigamiErc4626WithRewardsManager`:

```
    function totalAssets() external view override returns (uint256 totalManagedAssets) {
        // @audit depositedAssets() also includes assets in the contract's balance that hasn't been yet
        ↪  deposited
>>>     return _totalAssets(depositedAssets());
    }


    function depositedAssets() public view override returns (uint256) {
        // @audit unallocatedAssets includes assets that are still in the contract's balance
>>>     return unallocatedAssets() + underlyingVault.previewRedeem(
            underlyingVault.balanceOf(address(this))
        );
    }


    function unallocatedAssets() public view override returns (uint256 amount) {
        // @audit these assets haven't been supplied yet to the underlying as they are still in the
        ↪  contracts's balance
>>>     amount = _asset.balanceOf(address(this));
        (amount,) = amount.splitSubtractBps(_performanceFeeBps, OrigamiMath.Rounding.ROUND_DOWN);
    }


    function _totalAssets(uint256 totalStaked) internal view returns (uint256 totalManagedAssets) {
        // Total assets = staked amount - unvested rewards - any future period (yet to start vesting)
        ↪  reserves
        (, uint256 unvested) = _vestingStatus();
        uint256 totalUnvested = unvested + futureVestingReserves;
        // Will have more staked than what is unvested, but floor to 0 just in case
        // @audit the assets in the contract's balance are counted as totalStaked, but when deposited,
        ↪  they are counted as futureVestingReserves
        //  therefore creating a jump in share price
        unchecked {
            return totalStaked > totalUnvested ? totalStaked - totalUnvested : 0;
        }
    }
```

So, inside `_totalAssets()`, we have that:

- If some assets are in the contract balance but not deposited yet, they are counted as `totalStaked`, because they are returned as part of `depositedAssets()`. Therefore, **add** to the `totalManagedAssets`.
- As soon as these assets are deposited by calling `manager.deposit(amount)` with the `amount` of assets in the balance, they count as `totalUnvested` (part of futureVestingReserves). Therefore, they are **subtracted** from the `totalStaked` when returning `totalManagedAssets`.
- Therefore, the action of depositing those assets (with `reinvest()`) will decrease the output of `_totalAssets()` by twice the `assets` in the manager's balance, making the share price drop.

Calling `reinvest()` with an `amount` of `assets` in the contract's balance will drop the `_totalAssets()` by `2 * amount`. This drop in total assets translates into a drop in the vault share price.

The issue only appears when there is a delay between the moment of the assets arriving in the balance, and the call to `reinvest()`. The price drops when calling reinvest, so if it is called in the same transaction as when the assets arrive in the balance, there is no issue.

Impact

Assets held in the manager contract will be considered already vested assets, and when `reinvest()` is called, they will be considered not vested. This creates a drop in the share price between before and after `reinvest()`, which it shouldn't.

Luckily, that is the case most of the time as:

- When users deposit to the vault, these are deposited immediately in the underlying vault, without spending any time in the manager
- When the rewards are swapped in the swapper and transferred to the manager, the swapperWithCallback will immediately reinvest, making this edge case invalid.

Therefore, in normal conditions, this scenario should not happen. This issue only manifests if another entity deposits assets to the manager without calling `reinvest()`.

Furthermore, exploiting this with a donation of assets seems implausible, as the profit from front-running the share price drop would hardly ever compensate for the loss of donating the tokens.

Fix

The assets in the manager balance should not be included in `depositedAssets()` until they have been deposited and checkpointed.

```
    function depositedAssets() public view override returns (uint256) {
-       return unallocatedAssets() + underlyingVault.previewRedeem(
+       return underlyingVault.previewRedeem(
            underlyingVault.balanceOf(address(this))
        );
    }
```

Proof of code

The following test can be added at the end of :

`origami/apps/protocol/test/foundry/unit/investments/erc4626/OrigamiErc4626WithRewardsVault.t.sol`:

Note, I added an import `import {console} from "forge-std/console.sol";`.

```
    function test_sharePriceDropOnDeposit_ifAssetsInBalance() public {
        // normal deposit by a normal user
        deposit(alice, 100e18);

        // price pre-donation is 999999999999999989
        console.log("Share price before donation: %s", vault.convertToAssets(1e18));

        // setup the issue: donate some assets to the manager
        uint256 donatedAmount = 10e18;
        deal(address(USDS), address(manager), donatedAmount, true);
        console.log("Share price after donation increases (wrongly): %s", vault.convertToAssets(1e18));

        // the price increases with the donation: 1098901098901098890
        uint256 sharePricePreReinvest = vault.convertToAssets(1e18);

        // now calling deposit() will trigger the
        manager.reinvest();

        // this will revert, because the share price drops back to 999999999999999989 (lower than
        ↪  pre-reinvest)
        console.log("Share price after donation drops: %s", vault.convertToAssets(1e18));
        assertGe(vault.convertToAssets(1e18), sharePricePreReinvest, "Share price should not drop below
        ↪  pre-reinvest value");
    }
```

The above test reverts with the message:

```
      [0] VM::assertGe(999999999999999989 [9.999e17], 1098901098901098890 [1.098e18], "Share price should
    ↪  not drop below pre-reinvest value") [staticcall]
        ← [Return]
     ← [Stop]
```

Response:

Fixed in commit d2099b5943b698c6d8a77840cf96854b5432681d

8.2 Low Risk

8.2.1 [L1] - Underlying Vault Withdrawal Fee Not Passed to User

Location:

`OrigamiErc4626WithRewardsManager.withdraw()`

Description:

When the underlying vault has a withdrawal fee, the user withdrawing doesn't perceive it directly, because the manager transfers the assets as a whole. Meanwhile, in the same transaction the underlying would have burned some shares, reducing the share supply of the remaining vault shareholders. In practical terms, the fee is distributed among the remaining depositors.

The share price would decrease because more shares are burned as fees, so remaining depositors effectively pay the fee indirectly through a decreased share price.

Impact:

Users withdrawing don't see the actual cost of withdrawal fees, while remaining users bear the cost through dilution. This creates an unfair cost distribution where the person initiating the withdrawal doesn't bear the full cost of their action.

Mitigation:

Consider implementing a mechanism calculate the amount of shares from the user should be withdrawn, proportional to the amount to of shares burned by the underlying relative to the total supply.

Alternatively, make sure this is only used with vaults that don't apply a withdraw fee.

Code Context:

```
function withdraw(uint256 assetsAmount, address receiver) external override onlyVault returns (uint256)
↪  {
    if (assetsAmount > 0) {
        emit AssetWithdrawn(assetsAmount);
        underlyingVault.withdraw(assetsAmount, receiver, address(this));
    }
    // ...
}
```

Response:

Acknowledged, this will only be used with vaults with 0 withdrawal fee.

8.2.2 [L2] - Manager switch fails when underlying vault has withdrawal limits

Location:

`setManager()`

Description:

The `setManager` function attempts to withdraw all assets from the old manager at once using `oldManager.totalAssets()` as the withdrawal amount. If the underlying vault has withdrawal limits or liquidity constraints that prevent withdrawing the full amount in a single transaction, the `withdraw()` call will revert, making it impossible to switch managers.

Impact:

For vaults with withdrawal limits or liquidity constraints, it becomes impossible to switch managers.

Code Context:

```
function setManager(address newManagerAddress, uint256 minNewAssets) external virtual override
↪  onlyElevatedAccess {
    // ...
    if (address(oldManager) != address(0)) {
        uint256 existingAssets = oldManager.totalAssets();
        uint256 withdrawnAssets = oldManager.withdraw(existingAssets, newManagerAddress);
        newManager.deposit(withdrawnAssets);
        // ...
    }
    // ...
}
```

Response: Acknowledged

The team only intends to use this feature with vaults where the full balance can be withdrawn at once.

8.3 Gas Optimizations

Use _maxMint() instead of maxMint() in mint function **Location**:

`OrigamiErc4626.mint()`

Description:

The `mint()` calls `maxMint(receiver)` which is a public view function that internally calls `_maxMint()`. This creates unnecessary gas overhead by making an external function call when the internal function could be called directly.

Mitigation:

Replace `maxMint(receiver)` with `_maxMint()` inside `mint()`.

Use >= comparison for early return in _maxDeposit **Location**:

`OrigamiErc4626._maxDeposit()`

Description:

The function `_maxDeposit` uses `if (_totalSupply > maxTotalSupply_) return 0;`. This could be optimized to use >= to avoid unnecessary gas consumption in subsequent operations when `_totalSupply` equals `maxTotalSupply_`.

8.4 Informationals

Function naming suggestion improvement for `_isVaultAsset()` **Location**:

OrigamiErc4626WithRewardsManager._isVaultAsset()

Description:

The function `_isVaultAsset()` should be named `_isAssetProtected()` or similar, as an ERC4626 can only have one "asset". The current name is misleading since it checks for both the base asset and the underlying vault token, which are protected from recovery rather than being vault assets.

Mitigation:

Consider renaming the function to better reflect its purpose, such as `_isProtectedToken()` or `_isAssetProtected()`.

Erc4626 standard consistency: `_maxDeposit()` should use `ROUND_DOWN` **Location**:

`OrigamiErc4626._maxDeposit()`

Description:

The `_maxDeposit` function uses `ROUND_UP` when converting to assets. However, for consistency with ERC4626 standards, `_maxDeposit` should round down to ensure it doesn't report a value higher than the actual maximum deposit amount possible.

Impact:

This creates an inconsistency between the reported maximum deposit value and the actual maximum assets that can be deposited. However, there is no real economic impact since the actual `deposit()` function uses `_previewDeposit()` which rounds down correctly.

Mitigation:

Change the rounding direction from `ROUND_UP` to `ROUND_DOWN` in the `_convertToAssets` call within `_maxDeposit`.

```
function _maxDeposit(uint256 feeBps) internal view returns (uint256 maxAssets) {
    // ...
    return _convertToAssets(
        availableShares.inverseSubtractBps(feeBps, OrigamiMath.Rounding.ROUND_UP),
        OrigamiMath.Rounding.ROUND_UP
    );
}
```

`OrigamiDelegated4626Vault.maxMint()` function has minor ERC4626 standard inconsistency regarding paused state   **Location**:

`OrigamiDelegated4626Vault.maxMint()`

Description:

There is a small inconsistency with the ERC4626 standard where `maxMint` should return 0 when deposits are paused, but the current implementation may not handle this correctly. The `_preview` functions should continue to work normally even when paused, but `maxMint` specifically should return 0 to indicate no shares can be minted.

Impact:

Minor deviation from ERC4626 standard expectations. External integrations that rely on `maxMint` returning 0 when deposits are paused might behave unexpectedly, though this is unlikely to cause significant issues in practice.

Mitigation:

Return 0 on `maxMint()` when paused. The same applies to `maxDeposit()`, `maxRedeem()` and `maxWithdraw()`

Morpho Claim Transaction Can Be Front-Run   **Location**:

OrigamiErc4626WithRewardsManager.morphoClaim()

Description:

The `morphoClaim()` transaction can be front-run with only one of the tokens, leaving the rest of the tokens unclaimed.

There is no economic incentive for the attacker beyond griefing, and even so, it doesn't destroy wealth, only delays the claiming of rewards. The likelihood of the attack is very low.

Impact:

Delay of the rewards claim from Morpho. This is purely informational, as it is not worth any code changes.

Code Context:

```
function morphoClaim(
    address[] calldata tokens,
    uint256[] calldata amounts,
    bytes32[][] calldata proofs
) external override {
    // ...
    for (uint256 i; i < tokens.length; ++i) {
        morphoRewardsDistributor.claim(address(this), tokens[i], amounts[i], proofs[i]);
    }
    // ...
}
```

VestingReserves not updated when pendingReserves is zero   **Location**:

`OrigamiVestingReserves._checkpointPendingReserves()`

Description:

When pendingReserves is zero, the function returns early without updating vestingReserves. This means that once a vesting period is complete and no new reserves are pending, the old vestingReserves value remains set instead of being cleared to zero.

Impact:

There is no economic impact because the _vestingStatus() function correctly returns zero for unvested reserves when the vesting period is complete, and the totalAssets calculation remains unaffected. The issue is purely one of state consistency between what _vestingStatus() and vestingReserves return.

Mitigation:

Consider setting vestingReserves = 0 when a vesting period is complete and no new reserves are pending. Only skip updates once both futureVestingReserves and vestingReserves are zero.

Code Context:

```
function _checkpointPendingReserves(uint256 amountReinvested) internal {
    uint128 pendingReserves = (futureVestingReserves + amountReinvested).encodeUInt128();

    // @audit if futureVestingReserves is zero, vestingReserves is not updated at the end of the vesting
    ↪  period
    if (pendingReserves == 0) return;

    // ...
    if (secsSinceLastCheckpoint < reservesVestingDuration) {
        futureVestingReserves = pendingReserves;
    } else {
        vestingReserves = pendingReserves;
        lastVestingCheckpoint = currentTime;
        futureVestingReserves = 0;
    }
}
```