Electisec Origami hOHM Review

Review Resources:

- Documentation

Auditors:

- Adriro
- Panda

Table of Contents

Review Summary

Origami hOHM

Origami hOHM is a cross-chain DeFi protocol that tokenizes leveraged OHM positions by maximizing borrowing against gOHM collateral in the Olympus Cooler system to generate optimized yield.

The contracts of the Origami hOHM repository were reviewed over eight days. Two auditors performed the code review between February 27th and March 6th, 2025. The repository was under active development during the review, but the review was limited to commit 6b0e28eb43cd32f0bf61c600d9c9e561df6796de.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts
├── common
│   ├── OrigamiTokenizedBalanceSheetVault.sol
│   ├── access
│   │   └── OrigamiOftElevatedAccess.sol
│   ├── omnichain
│   │   ├── OrigamiOFT.sol
│   │   ├── OrigamiTeleportableToken.sol
│   │   └── OrigamiTokenTeleporter.sol
```

```
|   |── swappers
|   |   ├── OrigamiSwapperWithCallback.sol
├── investments
|   └── olympus
|       ├── OrigamiHOhmManager.sol
|       └── OrigamiHOhmVault.sol
└── libraries
    └── OlympusCoolerDelegation.sol
```

After the findings were presented to the Origami hOHM team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

Electisec and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. Electisec and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Origami hOHM and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Proper access control is implemented using the OrigamiElevatedAccess mixin.
Mathematics	Average	Some rounding issues were detected in the Tokenized Balance Sheet contract.
Complexity	Good	Complexity is handled with a modular architecture and well-balanced separation of concerns, enabling the potential reuse of the Tokenized Balance Sheet logic.
Libraries	Good	The contracts integrate with the OpenZeppelin and LayerZero V2 libraries.
Decentralization	Average	Part of the vault's lifecycle requires privileged access.
Code stability	Good	Contracts remained stable throughout the revision.
Documentation	Good	Documentation at the source code level is excellent. Additionally, high-level specs for the system were provided.
Monitoring	Good	Proper monitoring events for key operations are in place.
Testing and verification	Good	The codebase includes a unit and invariant test suite with good coverage.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
    - Findings that can improve the gas efficiency of the contracts.
- Informational

- Findings including recommendations and best practices.

Critical Findings

None.

High Findings

None.

Medium Findings

1. Medium - Share rounding is incorrect when the input token is a liability

When the input token is a liability, the functions `joinWithToken()` and `exitWithToken()` should round in the opposite direction.

Technical Details

The implementation of `joinWithToken()` rounds down the minted shares the user will receive. This behavior is correct when the input token is an asset since the vault should not mint more value than received, but it has the opposite effect when the input token is a liability.

Suppose we have the following scenario:

- `inputTokenBalance = 5`
- `totalSupply = 2`
- `tokenAmount = 4`
- `shares = joinWithToken(tokenAmount) = roundDown(tokenAmount * totalSupply / inputTokenBalance) = roundDown(4 * 2 / 5) = 1`.
- `exitWithShares(shares) = roundUp(shares * inputTokenBalance / totalSupply) = roundUp(1 * 5 / 2) = 3`

This means we got four tokens as a liability when we joined and deposited only three tokens when we exited.

The same happens with `exitWithToken()` as it always rounds up the number of required shares. This is fine when the input token is an asset since the vault should ensure the user gets burned at least the returned value. However, it has the opposite effect when the input token is a liability since the value is now being deposited into the vault.

Impact

Medium. Incorrect rounding leads to leaked value from the vault.

Recommendation

When the input token is a liability, round UP in `joinWithToken()` and round DOWN in `exitWithToken()`.

Developer Response

Fixed in 8b57aa6.

Low Findings

1. Low - Dust will be left in the vault contract when teleporting tokens

The vault pulls the amount in local decimals, but the teleporter pulls tokens, considering the shared decimals, generating a leftover in the vault contract.

Technical Details

The implementation of OrigamiTeleportableToken.sol transfer the bridged tokens from the caller using the amount in local decimals:

```
82:         uint256 amount = sendParam.amountLD;
83:         IERC20(address(this)).safeTransferFrom(msg.sender, address(this),
amount);
```

However, OFT tokens have a concept of shared decimals, usually six by default, which is considered while bridging tokens by removing any potential dust from the given amount to account for the decimal precision difference.

This means that OrigamiTeleportableToken.sol will pull the amount with full precision. Still, OrigamiTokenTeleporter.sol will clean the decimal difference from the amount, creating a leftover in the vault contract of up to `1e12` in each operation.

Impact

Low. Small amounts of hOHM will be accumulated and locked in the vault contract.

Recommendation

Apply the same routine to remove the dust from the collected token amount in `OrigamiTeleportableToken::send()`.

Developer Response

Fixed in 48b8042.

2. Low - Incorrect rounding in max functions

An incorrect rounding direction in `maxJoinWithToken()` and `maxExitWithToken()` allows users to exceed the intended maximum limit.

Technical Details

Given a specific available share capacity, the implementation of `_maxJoinWithToken()` calculates the max amount of tokens using `_convertSharesToOneToken()` by rounding up if the token is an asset.

```
488:     function _maxJoinWithToken(
489:         _Cache memory cache,
490:         uint256 feeBps
491:     ) internal virtual view returns (uint256 maxTokens) {
492:         // If the token balance of the requested token is zero then cannot join
493:         if (cache.inputTokenBalance == 0) return 0;
494:
495:         // If this is an unknown asset then return zero
```

```
496:         (bool inputIsAsset, bool inputIsLiability) =
isBalanceSheetToken(cache.inputTokenAddress);
497:         if (!(inputIsAsset || inputIsLiability)) return 0;
498:
499:         uint256 maxTotalSupply_ = maxTotalSupply();
500:         if (maxTotalSupply_ == type(uint256).max) return maxTotalSupply_;
501:
502:         uint256 availableShares = _availableSharesCapacity(cache.totalSupply,
maxTotalSupply_);
503:
504:         // When calculating the max assets or liabilities amount, if the input
token is:
505:         //   - an asset: ROUND_UP (assets are pulled from the caller)
506:         //   - a liability: ROUND_DOWN (liabilities sent to recipient)
507:         // Shares are rounded down within previewJoinWithToken, so can be
rounded up when determining
508:         // the max here.
509:         return _convertSharesToOneToken({
510:             shares: availableShares.inverseSubtractBps(feeBps,
OrigamiMath.Rounding.ROUND_UP),
511:             cache: cache,
512:             rounding: inputIsAsset ? OrigamiMath.Rounding.ROUND_UP :
OrigamiMath.Rounding.ROUND_DOWN
513:         });
514:     }
```

```
845:     function _convertSharesToOneToken(
846:         uint256 shares,
847:         _Cache memory cache,
848:         OrigamiMath.Rounding rounding
849:     ) private pure returns (uint256) {
850:         // An initial seed deposit is required first.
851:         // In the case of a new asset added to an existing vault, a donation of
tokens
852:         // must be added to the vault first.
853:         return cache.inputTokenBalance == 0
854:             ? 0
855:             : shares.mulDiv(cache.inputTokenBalance, cache.totalSupply,
```

```
rounding);
856:     }
```

By rounding up on this calculation, the actual capacity would be exceeded. Example:

- `inputTokenBalance = 2`
- `totalSupply = 8`
- `availableShares = 1`
- `maxTokens = roundUp(1 * 2 / 8) = 1`.

Plugging this `maxTokens` amount in `previewJoinWithToken()` or `joinWithToken()` would yield `shares = roundDown(maxTokens * totalSupply / inputTokenBalance) = roundDown(1 * 8 / 2) = 4`. This means we can mint four shares while the capacity was just 1.

The same happens in `maxExitWithToken()` but for liabilities. The implementation rounds up the calculation when the token is one of the vault's liabilities.

```
595:     function _maxExitWithToken(
596:         _Cache memory cache,
597:         address sharesOwner,
598:         uint256 feeBps
599:     ) internal virtual view returns (uint256 maxTokens) {
600:         // If this is an unknown asset then return zero
601:         (bool inputIsAsset, bool inputIsLiability) =
isBalanceSheetToken(cache.inputTokenAddress);
602:         if (!(inputIsAsset || inputIsLiability)) return 0;
603:
604:         // Special case for address(0), unlimited
605:         if (sharesOwner == address(0)) return type(uint256).max;
606:
607:         uint256 shares = balanceOf(sharesOwner);
608:
609:         // When calculating the max assets or liabilities amount, if the input
token is:
610:         //   - an asset: ROUND_DOWN (assets are sent to the recipient)
611:         //   - a liability: ROUND_DOWN (liabilities are pulled from the caller)
612:         // Shares are rounded up within previewExitWithToken, so can be rounded
```

```
down when determining
613:         // the max here.
614:         (shares,) = shares.splitSubtractBps(feeBps,
OrigamiMath.Rounding.ROUND_DOWN);
615:         uint256 maxFromShares = _convertSharesToOneToken({
616:             shares: shares,
617:             cache: cache,
618:             rounding: inputIsAsset ? OrigamiMath.Rounding.ROUND_DOWN :
OrigamiMath.Rounding.ROUND_UP
619:         });
620:
621:         // Return the minimum of the available balance of the requested token
in the balance sheet
622:         // and the derived amount from the available shares
623:         return maxFromShares < cache.inputTokenBalance ? maxFromShares :
cache.inputTokenBalance;
624:     }
```

Impact

Low. Maximum amounts are overestimated.

Recommendation

`maxJoinWithToken()` or `maxExitWithToken()` should always round down for assets and liabilities.

Developer Response

Fixed in 8b57aa6.

Gas Saving Findings

None.

Informational Findings

1. Informational - Overflow risk in uint256 to int256 conversion

Technical Details

In the `OlympusCoolerDelegation` library, there is a potential integer overflow risk when converting from uint256 to int256 in the `_syncDelegationRequest` function:

```
int256 delta = int256(newDelegationAmount) - int256(existingDelegationAmount);
```

This is unlikely to be problematic in the current Olympus implementation, as the gOHM token amounts will never approach these extreme values. However, as this library could be reused in other contexts with different tokens, it represents a potential vulnerability if used with tokens that have very large supplies or amounts.

Impact

Informational

Recommendation

To future-proof the library against potential overflows when used in different contexts, consider implementing safe casting:

```
require(newDelegationAmount <= type(int256).max, "Amount exceeds int256 max");
require(existingDelegationAmount <= type(int256).max, "Amount exceeds int256 max");
int256 delta = int256(newDelegationAmount) - int256(existingDelegationAmount);
```

Developer Response

Fixed in c517116.

2. Informational - Pauses are not considered in max functions

The maximum functions ignore the pause state, potentially returning a positive capacity when the functionality is disabled.

Technical Details

- `maxJoinWithToken()`
- `maxJoinWithShares()`

- `maxExitWithToken()`
- `maxExitWithShares()`

Impact

Informational.

Recommendation

Return zero if joins or exits are paused.

Developer Response

Fixed in 3f7476d.

3. Informational - The last shareholder is charged an exit fee

The exit fee is applied when the last set of shares is redeemed, leaving a positive balance of assets and liabilities in the vault despite a final supply of zero.

Technical Details

The `exitWithToken` and `exitWithShares()` functions charge an exit fee which applies a cut to the returned assets and liabilities, increasing the value of other shareholders.

This exit fee is also charged when the last set of shares is redeemed, leaving a non-zero balance of tokens when the share supply is zero.

Impact

Informational.

Recommendation

Skip the exit fee when redeeming the total supply of shares.

Developer Response

Acknowledged. Not really an issue. When the vault is being unwound, the exit fee will be set to zero via `OrigamiHOhmManager::setExitFees()`.

4. Informational - Typos

Technical Details

```
File: contracts/common/OrigamiTokenizedBalanceSheetVault.sol

// @audit: initally should be initially
39: /// It is initally set to zero, and first set within seed()

// @audit: denomintor should be denominator
773: // denomintor is zero
```

OrigamiTokenizedBalanceSheetVault.sol#L39, OrigamiTokenizedBalanceSheetVault.sol#L773

```
File: contracts/investments/OrigamiInvestmentVault.sol

// @audit: depoyed should be deployed
57: * @notice Track the depoyed version of this contract.

// @audit: underyling should be underlying
65: * @dev For an investment vault, this is the underyling reserve token

// @audit: Unforunately should be Unfortunately
105: /// Unforunately needs to copy the input array as this is memory defined
storage.
```

OrigamiInvestmentVault.sol#L57, OrigamiInvestmentVault.sol#L65, OrigamiInvestmentVault.sol#L105

Impact

Informational.

Recommendation

Fix the typos.

Developer Response

Fixed in 311e156.

5. Informational - Unused import

The identifier is imported but never used within the file.

Technical Details

```
File: contracts/common/OrigamiTokenizedBalanceSheetVault.sol

7: import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

OrigamiTokenizedBalanceSheetVault.sol#L7

Impact

Informational.

Recommendation

Remove the not used import.

Developer Response

Fixed in 08e9579.

6. Informational - Unused errors present

Unused error is defined and can be removed.

Technical Details

```
File: contracts/interfaces/common/IOrigamiTokenizedBalanceSheetVault.sol

19: error InvalidAsset();

36: error ERC2612ExpiredSignature(uint256 deadline);

39: error ERC2612InvalidSigner(address signer, address tokensOwner);
```

OrigamiTokenizedBalanceSheetVault.sol#L19, IOrigamiTokenizedBalanceSheetVault.sol#L36, IOrigamiTokenizedBalanceSheetVault.sol#L39

Impact

Informational.

Recommendation

Remove the unused errors.

Developer Response

Fixed in 20ee82d.

7. Informational - Max deposit limit is ignored when syncing savings

The implementation of `_syncSavings()` caps withdrawals using `ERC4626::maxWithdraw()` but ignores any potential limit while depositing.

Technical Details

```
543:     function _syncSavings(uint256 requiredDebtTokenBalance) private {
544:         IERC4626 _savingsVault = debtTokenSavingsVault;
545:         if (address(_savingsVault) == address(0)) return;
546:
547:         uint256 _debtTokenBalance = debtToken.balanceOf(address(this));
548:         if (_debtTokenBalance > requiredDebtTokenBalance) {
549:             // deposit any surplus into savings
550:             _savingsVault.deposit(_debtTokenBalance - requiredDebtTokenBalance,
address(this));
551:         } else {
552:             // withdraw deficit from savings. Cap to the max amount which can
be withdrawn.
553:             uint256 delta = requiredDebtTokenBalance - _debtTokenBalance;
554:             uint256 maxWithdraw = _savingsVault.maxWithdraw(address(this));
555:             if (delta > maxWithdraw) delta = maxWithdraw;
556:             if (delta > 0) {
557:                 _savingsVault.withdraw(delta, address(this), address(this));
558:             }
559:         }
560:     }
```

Impact

Informational.

Recommendation

Limit the deposit to the result of `ERC4626::maxDeposit()`.

```
        if (_debtTokenBalance > requiredDebtTokenBalance) {
            // deposit any surplus into savings
+           uint256 delta = _debtTokenBalance - requiredDebtTokenBalance;
+           uint256 maxDeposit = _savingsVault.maxDeposit(address(this));
+           if (delta > maxDeposit) delta = maxDeposit;
+           if (delta > 0) {
-               _savingsVault.deposit(_debtTokenBalance - requiredDebtTokenBalance,
address(this));
+               _savingsVault.deposit(delta, address(this));
+           }
        } else {
```

Developer Response

Fixed in e711b13.

8. Informational - Potential denial of service when hOHM total supply gets reduced to zero

The `_convertSharesToCollateral()` function reverts when the total supply is zero, leading to a denial of service if the last hOHM holder exits the vault.

Technical Details

The `_convertSharesToCollateral()` function adjusts the gOHM delegation amount for an account whenever the account's hOHM balance changes.

```
564:     function _convertSharesToCollateral(
565:         uint256 shares,
566:         uint256 totalCollateral,
567:         uint256 totalSupply,
568:         bool applyMinDelegationAmount
569:     ) private pure returns (uint256 collateral) {
```

```
570:         if (totalSupply == 0) revert CommonEventsAndErrors.ExpectedNonZero();
571:         if (shares > totalSupply) revert CommonEventsAndErrors.InvalidParam();
572:         collateral = totalCollateral.mulDiv(shares, totalSupply,
OrigamiMath.Rounding.ROUND_DOWN);
573:         if (applyMinDelegationAmount && collateral < MIN_DELEGATION_AMOUNT)
collateral = 0;
574:     }
```

Line 570 reverts the transaction if the total supply is zero. This should be fine as long the vault is not empty, but will cause a denial of service when the last holder of hOHM tries to exit their position.

Impact

Informational.

Recommendation

Return zero if the total supply is zero.

```
function _convertSharesToCollateral(
        uint256 shares,
        uint256 totalCollateral,
        uint256 totalSupply,
        bool applyMinDelegationAmount
    ) private pure returns (uint256 collateral) {
-       if (totalSupply == 0) revert CommonEventsAndErrors.ExpectedNonZero();
+       if (totalSupply == 0) return 0;
```

Developer Response

Fixed in d5c9636.

Final remarks

The new Origami hOHM protocol features an innovative vault design that extends beyond traditional single-asset vaults like ERC4626. This new vault includes multiple different assets

and represents potential liabilities, effectively forming a balance sheet. The resulting equity can then be used to value the associated token.

The inherent complexity of managing an interface that needs to deal with both assets and debt led to some rounding issues in the implementation, which the Origami team promptly addressed.

The auditors value the developers' dedication to providing a well-designed and structured codebase, with great attention to documentation and testing.