Guardefy

hOHM Vault

Audit Report

prepared by Pavel Anokhin (@panprog)

March 7, 2025

Version: 2

Confidential

Guardefy

Contents

1. About Guardefy and @panprog

Pavel Anokhin or **@panprog**, doing business as Guardefy, is an independent smart contract security researcher with a track record of finding numerous issues in audit contests, bugs bounties and private solo audits. His public findings and results are available at the following link:

https://audits.sherlock.xyz/watson/panprog

2. Disclaimer

Smart contract audit is a time, resource and expertise bound effort which doesn't guarantee 100% security. While every effort is put into finding as many security issues as possible, there is no guarantee that all vulnerabilities are detected nor that the code is secure from all possible attacks. Additional security audits, bugs bounty programs and on-chain monitoring are strongly advised.

This security audit report is based on the specific commit and version of the code provided. Any modifications in the code after the specified commit may introduce new issues not present in the report.

3. Scope of the audit

The code at the following link was reviewed:

https://github.com/TempleDAO/origami

commit hash: 2eb986428d755ec730398af50503be936adcb312

Files in scope:

```
./contracts/common/OrigamiTokenizedBalanceSheetVault.sol
./contracts/common/access/OrigamiOftElevatedAccess.sol
./contracts/common/omnichain/OrigamiOFT.sol
./contracts/common/omnichain/OrigamiTeleportableToken.sol
./contracts/common/omnichain/OrigamiTokenTeleporter.sol
./contracts/common/swappers/OrigamiSwapperWithCallback.sol
./contracts/investments/olympus/OrigamiHOhmManager.sol
./contracts/investments/olympus/OrigamiHOhmVault.sol
./contracts/libraries/OlympusCoolerDelegation.sol
```

4. Audit Timeline

Audit start: February 23, 2025

Audit report delivered: March 5, 2025

Fixes reviewed: March 7, 2025

5. Findings

5.1. High severity findings

None found.

5.2. Medium severity findings

5.2.1. [M-1] `OrigamiTokenizedBalanceSheetVault` reverts in some cases when trying to `exitWithToken` with amount of token equal to `maxExitWithToken`.

When vault liability token balance is much smaller than totalSupply, multiple possible share amounts convert to the same liability token amount:

```
totalSupply = 100, assets balance = 2, liabilities balance = 2
0 shares -> 0 assets, 0 liabilities
1 shares -> 0 assets, 1 liabilities
2 shares -> 0 assets, 1 liabilities
...
49 shares -> 0 assets, 1 liabilities
50 shares -> 1 assets, 1 liabilities
51 shares -> 1 assets, 2 liabilities
..
99 shares -> 1 assets, 2 liabilities
100 shares -> 2 assets, 2 liabilities
```

When doing inverse calculation (finding share amount from liabilities amount), the following formula is used:

```
tokenAmount.mulDiv(cache.totalSupply, cache.inputTokenBalance, rounding);
```

where rounding for shares is up:

```
        (sharesAfterFees, assets, liabilities) = _convertOneTokenToSharesAndTokens({
            tokenAmount: tokenAmount,
            cache: cache,
            sharesRounding: OrigamiMath.Rounding.ROUND_UP,
            assetsRounding: OrigamiMath.Rounding.ROUND_DOWN,
            liabilitiesRounding: OrigamiMath.Rounding.ROUND_UP
        });
```

This means that max amount of shares is chosen if multiple values are possible. For example, for liability amount of 1, `shares = 1*100/2 = 50`, for liablity amount of 2, `shares = 2*100/2 = 100`.

When calculating `maxExitWithToken`, the liabilities amount is rounded up:

```
        uint256 maxFromShares = _convertSharesToOneToken({
            shares: shares,
            cache: cache,
            rounding: inputIsAsset ? OrigamiMath.Rounding.ROUND_DOWN :
OrigamiMath.Rounding.ROUND_UP
        });
```

This means that any shares amount between 1 and 49 will round up to liability = 1, and any shares amount between 51 and 99 will round up to liability = 2.

The issue is that if user tries to `exitWithToken` with liability amount returned by `maxExitWithToken`, most of the time the call will revert trying to burn more shares than the user has. For example:

- user has 49 shares
- `maxExitWithToken` returns liability = 1
- user tries to `exitWithToken` with liability = 1
- internal shares calculated will be 50
- call reverts trying to burn 50 shares while the user only has 49 shares.

Likelihood

High. The issue happens any time `totalSupply` is at least twice the total liability, which is quite likely. Then most attempts to exit with token amount returned by `maxExitWithToken` will revert.

Impact

All attempts to exit with such token amount unexpectedly revert. The impact can range from gas loss and re-try with smaller token amount (causing user confusion) to funds lock/loss of integrating protocols expecting this function to always work.

Possible mitigation

The best possible solution is to choose the smallest amount of shares in the case described. For example, for liability = 1, the shares amount calculated should be 1, not 50. This way the user will always have enough shares balance for the given liability. The formula for shares will then be:

```
uint add = totalSupply > tokenBalance ? 1 : 0
shares = rounddown((inputAmount-add) * totalSupply / tokenBalance) + add
```

Alternative solution: keep `previewExitWithToken` as is, but modify `exitWithToken` to choose the minimum amount of shares from user balance and calculated amount. Downside is that `previewExitWithToken` returned amounts will differ from the actual `exitWithToken` amoutns.

Another alternative: round down number of liabilities from shares when calculating `maxExitWithToken`. The downside is that `exitWithShares` can result in max liability (e.g. 2) while `maxExitWithToken` will return a value less than that (e.g. 1).

Status: Fixed

Fix Review: maxExitWithToken rounds down the amount of token, including liability. This fixes the issue.

5.3. Low severity findings

5.3.1. [L-1] OrigamiTokenizedBalanceSheetVault incorrect rounding when calculating shares in _previewJoinWithToken and _previewExitWithToken.

Currently _previewJoinWithToken calculates the number of shares to mint to user rounding it down:

```
// When calculating the number of shares/assets/liabilities when JOINING
// given an input token amount:
//   - shares: ROUND_DOWN (number of shares minted to the recipient)
//   - assets: ROUND_UP (the assets which are pulled from the caller)
//   - liabilities: ROUND_DOWN (liabilities sent to recipient)
(shares, assets, liabilities) = _convertOneTokenToSharesAndTokens({
    tokenAmount: tokenAmount,
    cache: cache,
    sharesRounding: OrigamiMath.Rounding.ROUND_DOWN,
    assetsRounding: OrigamiMath.Rounding.ROUND_UP,
    liabilitiesRounding: OrigamiMath.Rounding.ROUND_DOWN
});
```

However, this is incorrect, because assets and liabilities should be treated differently:

- Vault's assets per share must not decrease after operation (as assets are added to the balance): this requires rounding shares down.
- Vault's liabilities per share must not increase after operation (as liabilities are subtracted from the balance): this requires rounding shares up.

Example when liabilities per share increase after the join:

```
totalSupply = 21, liabilities = 11
joinWithToken(liability: 1)
shares to mint = round_down(1 * 21 / 11) = 1
new totalSupply = 22, liabilities = 12
liability per share before the operation: 11/21 = 0.5238
liability per share after the operation: 12/22 = 0.5455 (increased)
```

In the same way, when exiting with token, currently the shares are always rounded up, but:

- Vault's assets per share must not decrease after operation: this requires rounding shares up.
- Vault's liabilities per share must not increase after operation: this requires rounding shares down.

While both assets and liabilities are affected by the rounding, since they can be orders of magnitude different from each other, 1 wei of rounding error in one of them might be worth much more or much less than 1 wei of rounding error in the other one, as such total vault value might decrease when rounded incorrectly.

Likelihood

High. Likely will happen almost always when joining or exiting with the liability token.

Impact

Low, dust decrease of value for all vault owners (in the orders of 1 wei of either asset or liability). Impact on integrations is unknown and can be both low and high.

Possible mitigation

When joining with token: round shares down for assets, round up for liabilities.

When exiting with token: round shares up for assets, round down for liabilities.

Status: Fixed.

Fix Review: Fixed as suggested. Recommend to fix comment for _previewExitWithToken: shares rounding described twice: correct and incorrect (always round up).

5.3.2. [L-2] OrigamiTokenizedBalanceSheetVault usage of amounts proportional to token amount leads to incorrect rounding in favor of the user instead of the vault or to large loss of user funds.

Currently `_previewJoinWithToken` and `_previewExitWithToken` calculate the number of the other tokens as following:

- First, shares amount is calculated from the token amount
- Second, all the other token amounts are calculated proportional to input token amount and input token vault balance.

```
// otherTokenAmount = inputTokenAmount * otherTokenBalance / inputTokenBalance
amounts[i] = inputTokenAmount.mulDiv(
    _getTokenBalance(tokenAddresses[i], cache),
    cache.inputTokenBalance,
    rounding
```

The issue here is that rounding of shares vs token amounts becomes disconnected. As such the resulting amount of tokens (other than input token) only loosely matches with the shares percentage and breaks rounding.

Example when liabilities per share increase after the join:

```
totalSupply = 21, assets = 11, liabilities = 11
joinWithToken(asset: 1)
shares to mint = round_down(1 * 21 / 11) = 1
liabilities to be sent to user = round_down(1 * 11 / 11) = 1
new totalSupply = 22, assets = 12, liabilities = 12
liability per share before the operation: 11/21 = 0.5238
liability per share after the operation: 12/22 = 0.5455 (increased)
```

Another issue here is that with low totalSupply but high token amount the user unfairly loses large amount of token. For example:

```
totalSupply = 2, assets = 1000, liabilities = 1000
exitWithToken(liability: 1)
shares to burn = round_up(1 * 2 / 1000) = 1
assets sent to user = round_down(1 * 1000 / 1000) = 1
new totalSupply = 1, assets = 999, liabilities = 999
user got: 1 asset for 1 share and 1 liability
if asset = 2 * liability, then 1 share = 500 * 2 – 500 = 500
but user only got 1 *2 – 1 = 1 of value for his 1 share
```

Likelihood

High. Very likely to happen almost always.

Impact

Low, dust amount losses, or higher amount losses in very rare situations of too low totalSupply.

Possible mitigation

Convert from token to shares, then calculate the other tokens amounts proportional to shares rather than input token.

Status: Fixed

Fix Review: Fixed as suggested.

5.3.3. [L-3] OrigamiHOhmVault: attacker can steal most of the surplus buffer excess when surplus buffer exceeds vault debt.

OrigamiHOhmVault currently uses the following formula to calculate amount of debt token (USDS) sent to or received from the user on joins or exits:

```
return _coolerDebtInDebtTokens > _surplus
    ? _coolerDebtInDebtTokens - _surplus
    : 0;
```

As can be seen from the code, if surplus buffer exceeds the vault debt, a debt token amount of 0 is returned regardless of surplus buffer size. This means that when surplus buffer >= vault debt, 1 hOHM = x gOHM, both on joins and exits (so a fixed ratio between hOHM and gOHM).

The issue, however, is that this excess of surplus buffer does have value for the vault holders, as once it's used to buyback, the hOHM share price will increase (surplus buffer reduced, but 1 hOHM will now be worth more than x gOHM with still no debt token added in. For example consider 2 situations:

```
1) 100 hOHM = 100 gOHM - 300 USDS debt + 300 USDS surplus
1 hOHM = 1 gOHM
2) 100 hOHM = 100 gOHM - 300 USDS debt + 600 USDS surplus
1 hOHM = 1 gOHM
```

In both situations you can join or exit 1 hOHM for 1 gOHM, but in the 2nd situation expectation is that the share price will sharply increase once surplus buffer is used to buyback. For example, 300 USDS is used to buyback 50 hOHM and burn them, then:

```
50 hOHM = 100 gOHM - 300 USDS debt + 300 USDS surplus
1 hOHM = 2 gOHM
```

Attack scenario

In situation 2 attacker can join the vault with max amount he can afford and shortly after buyback is executed, exit everything to basically "steal" this surplus buffer excess from the other vault users:

- Attacker joins in situation 2 with 900 hOHM for 900 gOHM, 2700 USDS is borrowed by the vault, all of it is added to surplus buffer:

```
1000 hOHM = 1000 gOHM - 3000 USDS debt + 3300 USDS surplus
```

- 300 USDS of surplus buffer is used to buyback 50 hOHM and burn:

```
950 hOHM = 1000 gOHM - 3000 USDS debt + 3000 USDS surplus
1 hOHM = 1.053 gOHM
```

- Attacker exits the vault with 900 hOHM, 9 hOHM are burnt as exit fee (1%), attacker gets 937.9 gOHM and leaving the vault with:

```
50 hOHM = 62.1 gOHM – 186.3 USDS debt + 186.3 USDS surplus
1 hOHM = 1.242 gOHM
```

As can be seen, if not for attacker, the honest vault users received 2 gOHM per 1 hOHM after the buyback.

However, attacker steals 37.9 gOHM in a short amount of time, leaving vault users with only 1.242 gOHM per 1 hOHM.

Likelihood

Very unlikely as surplus buffer is not supposed to be greater than debt. This situation can happen only if a large donation to surplus buffer is done by some user.

Additionally, the example above is intentionally extreme and unrealistic (huge donation, large surplus buffer is exchanged at once) to highlight the issue. In real life the likelihood of such attack is further reduced by:

- Risk for the attacker to lose instead of profit from it: the other users can join after the attacker, reducing attacker's profit, possibly up to a point where it's no longer profitable due to exit fee.
- Donation size (excess of surplus above debt). If it's small (e.g. 10 USDS in the example), the buyback will increase hOHM price only slightly, thus exit fee might make the attack unprofitable.
- Buyback has restrictions on the amount to exchange at once and time until next buyback. This will prolong the time needed for buyback to finish, reducing attacker's ROI. This, by itself, doesn't stop the attack, but makes it more risky as more users can join over extended time and wash out the attacker's profit, rendering attack useless.

That said, the issue still stands even without attacker: if a large donation is made so that surplus buffer becomes sufficiently larger than vault's debt, then the other honest users will see the expected fast increase of hOHM share price, so they will join the vault in expectation of outsized returns, thus washing out existing users profit from buybacks. These users might not necessarily be profitable at this due to exit fees, but they will still get some value from this and will reduce the profit of existing users.

Impact

Almost all excess of surplus buffer over vault debt is lost by existing vault users. This is in contrast to the other situations when existing users benefit immediately from the donations:

- if donation is done by donating gOHM or burning hOHM - all existing users benefit from it immediately (if they exit, they receive more)

- if donation is done by repaying the hOHM debt or donating to surplus buffer such that surplus buffer becomes larger than debt, all existing users do not immediately benefit from this excess of donation, rather it becomes a future expected profit of the vault, and thus new joiners can share this expected price increase (reducing the increase with it), which is unfair to existing users

Possible mitigation

Let the debt be both positive and negative, so that when surplus buffer exceeds vault debt, users have to pay both gOHM and USDS to join the vault, and will receive both gOHM and USDS when exiting the vault (to reflect true hOHM value).

Alternatively solution might be based on the fact that the attacker benefits only after the buyback. If the vault surplus is used in some other way beneficial to the protocol, the attacker will be unable to benefit from the buyback and might lose funds (due to vault exit fees). After all, users are not guaranteed to benefit fully from the random donations.

Status: Acknowledged

5.4. Informational severity findings

5.4.1. [I-1] OrigamiTokenizedBalanceSheetVault has areJoinsPaused and areExitsPaused which are ignored in actual join and exit functions.

The vault has specific functions to determine if joins and exits are paused, but they are never used, although it'd be very logical to revert joins / exits if they're true. Instead, this is pushed to implementations (hOHM vault reverts joins and exits in the manager, for example).

It might be better and more logical to revert _join and _exit directly in OrigamiTokenizedBalanceSheetVault or at least add comment to _joinPreMintHook and _exitPreBurnHook with explanation that it should revert when joins/exits are paused.

Status: Fixed

Fix review: Fixed as suggested (_join and _exit revert if areJoinsPaused / areExitPaused return true)

5.4.2. [I-2] Comments typos.

_availableSharesCapacity:

```
/// @dev Much much spare capacity under the maxTotalSupply is the vault currently
function _availableSharesCapacity(
```

5.4.3. [I-3] OrigamiHOhmVault: performance fee charged by the protocol when executing a buyback translates to very large fee from buyback profits.

OrigamiHOhmVault charges performance fee on all buybacks by burning this fee percentage from all hOHM amounts received from buybacks. Since surplus buffer is used for buybacks, this means that:

- Surplus buffer reduces by the **x** amount;
- At the same time, the value added to vault (via hOHM burn) is at least performance fee% less than **x** amount

While the default performance fee = 3.3% doesn't look large, it eats large percentage of the actual users profit from the buyback feature.

The simulation at the following link with settings more or less close to real life scenario:

https://docs.google.com/spreadsheets/d/1dV3qIJTsYcJ-dxSgRNPpuaFIrgBcv1_3I3AYrrpRgcY/edit?usp=sharing

According to simulation (numbers at the top):

- initial hOHM price = 1
- no buyback: 1.220869
- buyback 0%: 1.230197
- buyback 3.3%: 1.225915

- buyback 0.3%: 1.229807

As can be seen from the table, buyback feature brings very little extra profit to hOHM: just 0.76%, while the other features generate 22.09% profit

However if we think of this feature in isolation:

- profit if no fees: 0.76%

- profit if fees = 3.3%: 0.41% (45% protocol fee off the profit)

- profit if fees =0.3%: 0.73% (4% protocol fee off the profit)

The default protocol fees of 3.3% from each buyback transaction end up eating 45% fee off the users profit from the buyback feature.

That said, this only applies to the buyback feature in isolation: the feature doesn't generate much profit to begin with and default performance fee eats a lot of it. However, if the performance fee is compared against the whole hOHM protocol with all the features, then the performance fee is about 0.4% from hOHM price or 1.9% from the profit, which is quite cheap for comparable protocols.

The effect of share price decrease can be further increased by the attackers who might sandwitch the buyback transaction with exiting and joining vault again. This can only happen in extreme scenarios where buyback amount is comparable to vault TVL, which shouldn't happen in real life. In all the other situations exit fee of 1% will prevent it, since effect of single buyback to hOHM price is less than 1%.

Likelihood

Default performance fees of 3.3% will always be very high percentage of buyback feature profit. Sandwitch attack is only possible in extremely rare situations (buyback amount comparable to vault TVL).

Impact

No issue, just information on the impact of performance fee to user's profit.

Possible mitigation

A performance fee of 0.3% is more appropriate for the buyback feature in isolation (it will then be about 4% of users profit). However, as a fee for the whole protocol a fee of 3.3% is acceptable.

Status: Acknowledged