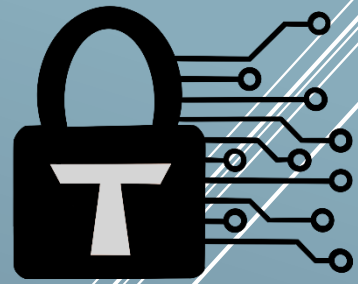


Trust Security

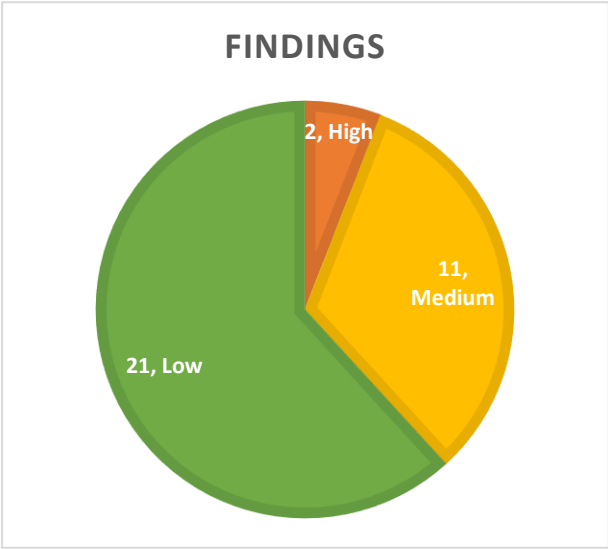


Smart Contract Audit

Olympus RBSv2

02/05/24

Executive summary



Category	Reserve
Audited file count	37
Auditor	Trust
Time period	08/01/24-14/02/24

Findings

Severity	Total	Open	Fixed	Acknowledged
High	2	-	2	-
Medium	11	-	6	5
Low	21	-	13	8

Centralization score



Signature



Digitally signed by Trust
Date: 2024.05.02 15:06:16 +03'00'

EXECUTIVE SUMMARY	1
DOCUMENT PROPERTIES	4
Versioning	4
Contact	4
INTRODUCTION	5
Scope	5
Repository details	6
About Trust Security	6
About the Auditors	7
Disclaimer	7
Methodology	7
QUALITATIVE ANALYSIS	8
FINDINGS	9
High severity findings	9
TRST-H-1 Heart may delete the wrong asset in removeMovingAverageAsset(), leading to pricing malfunctions	9
TRST-H-2 Anyone can prevent heartbeats, stagnating the system	10
Medium severity findings	11
TRST-M-1 The wall may not satisfy swaps due to inconsistency in TRSY approval	11
TRST-M-2 Precision loss due to divide before multiply in several functions	12
TRST-M-3 LBBO can be manipulated through the BLVaultSupply module	13
TRST-M-4 An attacker can deny protocol liveness by abusing deviation threshold	14
TRST-M-5 The harvest reward calculation will always yield zero reward, disincentivizing harvests	15
TRST-M-6 Bunni yield can be permanently stuck and uncollectable by the BunniManager	16
TRST-M-7 The protocol could face losses due to arbitrage through incorrect accounting of the Bunni holdings	19
TRST-M-8 The submodule observation array could be corrupted leading to wrong readings	20
TRST-M-9 The LBBO value will be incorrect after withdrawing from BunniSupply	21
TRST-M-10 LBBO can be manipulated over time through the BunniSupply which uses current reserves	21
TRST-M-11 LBBO can be manipulated over time through the BunniPrice which uses current reserves	22
Low severity findings	23
TRST-L-1 In the Operator the bond pricing would be incorrect if price has 36 digits	23
TRST-L-2 Property getters would return incorrect data when max age is the current timestamp	24
TRST-L-3 isDeviating() suffers from rounding issue, erring by up to 1bps	25

TRST-L-4 The ERC4626 pricer does not support tokens that comply with the spec and use a popular token mechanic	26
TRST-L-5 BunniManager does not assign categories for all groups when adding pool tokens.	27
TRST-L-6 Liquid backing calculation would be wrong if complex assets including OHM would be marked illiquid	28
TRST-L-7 thirtyDayOhmVolatility() does not use Bessel's correction	29
TRST-L-8 Appraiser miscalculates annual volatility by 0.1%	30
TRST-L-9 OlympusPriceV2 does not mark feed as unsuccessful if it returns a price of zero	31
TRST-L-10 addBunniToken() allows configuration of insufficient TWAP period	32
TRST-L-11 LBBO may be miscalculated due to the SiloSupply module	32
TRST-L-12 OlympusPriceV2 getCurrentPrice() may use a cached price	33
TRST-L-13 Reserve calculation may count GOHM as part of backing value	34
TRST-L-14 Tight coupling of modules and categories	36
TRST-L-15 The wall may be depleted sooner than intended in Operator	36
TRST-L-16 Leaked approval in the BunniManager	37
TRST-L-17 The Config contracts are not able to use all the necessary Module actions	38
TRST-L-18 Validation of heartbeat frequency is missing when distributor is updated	40
TRST-L-19 A metric that has been removed and added to tracking can be stale	41
TRST-L-20 Asset price can be updated twice per period	41
TRST-L-21 If BunniSupply contains positions with gOHM, LBBO calculation will be wrong	42
Additional recommendations	43
TRST-R-1 Docs improvements	43
TRST-R-2 Safety validations	43
TRST-R-3 Gas Optimizations	43
TRST-R-4 Readability	43
TRST-R-5 Coding style	44
TRST-R-6 Price feed strategy	44
TRST-R-7 Decoupling decimal assumptions	44
TRST-R-8 Unused variables	44
TRST-R-9 Improve reliability of the protocol	44
TRST-R-10 Improve versatility of BrickedSupply contract	45
TRST-R-11 Improve resistance to manipulation of the LBBO	45
TRST-R-12 Improve emission of events	45
TRST-R-13 Improve slippage check at BunniManager	46
TRST-R-14 Potentially trimming issues in observation calculations	46
TRST-R-15 Setting unchangeable variables as immutable	46
TRST-R-16 Add a staleness bit for moving average data in Appraiser	47
Centralization risks	48
TRST-CR-1 Certain permissioned roles have key privileges and must be trusted	48
Systemic risks	49
TRST-SR-1 Integration risks	49
TRST-SR-2 Configuration assumptions	49

Document properties

Versioning

Version	Date	Description
0.1	14/02/24	Client report
0.2	18/04/24	Mitigation review
0.3	02/05/24	Mitigation review #2

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- src/libraries/Deviation.sol
- src/libraries/QuickSort.sol
- src/libraries/UniswapV3/BunniHelper.sol
- src/libraries/UniswapV3/Oracle.sol
- src/libraries/UniswapV3/PoolLibrary.sol
- src/libraries/UniswapV3/Positions.sol
- src/modules/PRICE/submodules/feeds/BunniPrice.sol
- src/modules/PRICE/submodules/feeds/ChainlinkPriceFeeds.sol
- src/modules/PRICE/submodules/feeds/ERC4626Price.sol
- src/modules/PRICE/submodules/feeds/UniswapV2PoolTokenPrice.sol
- src/modules/PRICE/submodules/feeds/UniswapV3Price.sol
- src/modules/PRICE/submodules/strategies/SimplePriceFeedStrategy.sol
- src/modules/PRICE/OlympusPrice.v2.sol
- src/modules/PRICE/PRICE.v2.sol
- src/modules/SPPLY/submodules/BLVaultSupply.sol
- src/modules/SPPLY/submodules/BunniSupply.sol
- src/modules/SPPLY/submodules/IncurDebtSupply.sol
- src/modules/SPPLY/submodules/MigrationOffsetSupply.sol
- src/modules/SPPLY/submodules/SentimentArbSupply.sol
- src/modules/SPPLY/submodules/SiloArbSupply.sol
- src/modules/SPPLY/submodules/SiloSupply.sol
- src/modules/SPPLY/OlympusSupply.sol
- src/modules/SPPLY/SPPLY.v1.sol
- src/modules/TRSRY/OlympusTreasury.sol
- src/modules/TRSRY/TRSRY.v1.sol
- src/policies/OCA/interfaces/IAppraiser.sol
- src/policies/OCA/Appraiser.sol
- src/policies/OCA/PriceConfig.v2.sol
- src/policies/OCA/SupplyConfig.sol
- src/policies/OCA/TreasuryConfig.sol
- src/policies/RBS/interfaces/IHeart.sol
- src/policies/RBS/interfaces/IOperator.sol
- src/policies/RBS/Heart.sol
- src/policies/RBS/Operator.sol

- src/policies/UniswapV3/interfaces/IBunniManager.sol
- src/policies/UniswapV3/BunniManager.sol
- src/Submodules.sol

Repository details

- **Repository URL:** <https://github.com/OlympusDAO/bophades>
- **Commit hash:** 04791b8943240a55f13e43cbf013846c53c189a6
- **Fix review commit hashes:**
 - Issue 350 - 2c2dbbd822edeb6dfa0e4d8526c5e6c8b8aa0028
 - Issue 379 - 58ad7ed700e7ebf7cf2eaa5a8340e426a66fa800
 - Issue 367 - 09e8ac04107a613657d0bd52cdd4706e9baefdfa
 - Issue 351 - 280eb844617fa9c565de70fdb8f872134a2aee21
 - Issue 368 - cc200d775ee56236a3888039a5bd7c3f8759b5ae
 - Issue 304 - 1be317376a83f05846f321b69cecfef70aa855bd
 - Issue 317 - 046e2a82986cf6ddad12711272cbb35d1bbfcf88
 - Issue 320 - 0ac4fca0dda99c2d7f51e6c118bb2ace92818c71
 - Issue 321 - 101b569bd9b068dfb27daac51ff90bbabea38371
 - Issue 373 - a457c07a7f685efffe33fed5ed3252a20d15d513
 - Issue 355 - ddd88957bcd8a7844ed07c9576ba2dbbede99145
 - Issue 306 - 584fb9a77234bbcb57a40ebdf23b8699b974ccb2
 - Issue 308 - 9d17a364fd4d9090b7052b364d7af7e06c7e7b8e
 - Issue 311 - baabdb320c5a20298eee57b7ebe9a20547fc1a05
 - Issue 312 - 101b569bd9b068dfb27daac51ff90bbabea38371
 - Issue 314 - 2926972aba726ca75f75c600fc7307712cdd068d
 - Issue 315 - 482107b2e18dd2eb0356fef445df15e4ffd1b84d
 - Issue 309 - 721c245893edead63526682586db85d73a2ea3f8
 - Issue 318 - dde0f88cc50c81c23ea4216bf328a6b90b2b9f84
 - Issue 369 - b39e9a03881e0f3b1edc18e224d11ebb196fe04e
 - Issue 356 - 54a7e45b9b999b04410265adf89c3f3889f4dec6
 - Issue 332 - 704726d5ce531143c26858668d38a9e539bf63b4
 - Issue 322/323/325/328/329 - 135d6e535cabf658f022e5d139234a63da531a6f
- **Fix review #2 commit hashes:**
 - Issue 354 - b66c5e1e83b14a565e6864eac3b0b8c6a4c97744
 - Issue 380 - 078978968fa7b7632adedc8e3d0178debade1c5
 - Issue 352 - 1756d8c62c3eb5cabd1f9dfecadc5d678f226389
 - Issue 339 - ed1bc28bce56cedc4ca8a4015f02c226c05c10e4

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Good	The code is very well modularized to keep complexity manageable.
Documentation	Excellent	Project is mostly very well documented.
Best practices	Excellent	Project consistently adheres to industry standards.
Centralization risks	Moderate	The multisig and addresses of certain roles have far-reaching privileges.

Findings

High severity findings

TRST-H-1 Heart may delete the wrong asset in `removeMovingAverageAsset()`, leading to pricing malfunctions

- **Category:** Storage-memory confusion
- **Source:** Heart.sol
- **Status:** Fixed

Description

The admin may remove an asset from the heart monitoring by calling `removeMovingAverageAsset()`

```
function removeMovingAverageAsset(address asset_) external
onlyRole("heart_admin") {
    if (asset_ == address(0)) revert Heart_InvalidParams();
    // Cache the moving average assets to avoid multiple
    SLOADs
    address[] memory cachedMovingAverageAssets =
movingAverageAssets;
    uint256 assetsLen = cachedMovingAverageAssets.length;
    bool foundAsset = false;
    for (uint256 i = 0; i < assetsLen; i++) {
        if (cachedMovingAverageAssets[i] == asset_) {
            cachedMovingAverageAssets[i] =
cachedMovingAverageAssets[assetsLen - 1];
            movingAverageAssets.pop();
            foundAsset = true;
            break;
        }
    }
    if (!foundAsset) revert Heart_InvalidParams();
    emit MovingAverageAssetRemoved(asset_);
}
```

Note that the code always pops the last element in storage, whilst the last member is copied to replace the target asset. However the copying is done on the memory-cached version instead of on storage. As a result, it will not have an effect, and the last asset is the one logically deleted. This would cause the `beat()` to stop pricing the wrong asset, which likely would lead to malfunctioning of the LBBO functionality.

Recommended mitigation

Replace the value in the storage version of `movingAverageAssets`.

Team response

Fixed.

Mitigation review

The issue has been addressed as suggested.

TRST-H-2 Anyone can prevent heartbeats, stagnating the system

- **Category:** Access-control issues
- **Source:** Heart.sol
- **Status:** Fixed

Description

PR #351 introduces a check that the distributor's staking contract has the same frequency as the heartbeat frequency. Additionally, the **lastBeat** is updated to the start of the current epoch.

```
// Sync heartbeat with staking contract
(uint256 epochLength, , uint256 epochEnd, ) =
    IStaking(distributor.staking()).epoch();
if (frequency() != epochLength) revert
Heart_InvalidFrequency();
lastBeat = uint48(epochEnd - epochLength);
```

The code is ran in *configureDependencies()*, which in the OlympusDAO architecture is permissionless and only supposed to export list of dependencies.

The fact the function changes state makes it abusable by an unpermissioned attacker. As soon as a new epoch starts, they could call *configureDependencies()*, which would set the **lastBeat** to the current timestamp, *without* actually calling *beat()*. At this point, *beat()* cannot be called because not enough time elapsed since **lastBeat**.

Without heartbeats, the RBS mechanism cannot function, and most functions involving prices and Supply contracts would revert.

Recommended mitigation

Refrain from setting any variables in a *configureDependencies()* function.

Team response

Fixed.

Mitigation review

The issue was resolved by making *configureDependencies()* only callable by the Kernel, so only privileged roles could abuse the described behavior.

Medium severity findings

TRST-M-1 The wall may not satisfy swaps due to inconsistency in TRSY approval

- **Category:** Logical flaws
- **Source:** Operator.sol
- **Status:** Acknowledged

Description

When a wall is regenerated is the only time the Operator updates the approval for the ERC4626-wrapped reserve asset.

```
uint256 currentApproval = _wrappedReserve.previewRedeem(
    TRSY.withdrawApproval(address(this), _wrappedReserve)
);
unchecked {
    if (currentApproval < capacity) {
        TRSY.increaseWithdrawApproval(
            address(this),
            _wrappedReserve,
            _wrappedReserve.previewWithdraw(capacity -
currentApproval)
        );
    }
}
```

Note that **capacity** is in reserve tokens, and the code is converting it to the 4626 equivalent with *previewWithdraw()*. The issue is that this conversion ratio can change between the time of regeneration and time of swapping through the wall. For the DAI reserve asset, this rate is calculated in the SavingsDai contract in the code below:

```
function convertToAssets(uint256 shares) public view returns
(uint256) {
    uint256 rho = pot.rho();
    uint256 chi = (block.timestamp > rho) ? _rpow(pot.dsr(),
block.timestamp - rho) * pot.chi() / RAY : pot.chi();
    return shares * chi / RAY;
}
```

As can be seen, the ratio is fluctuating due to configuration parameters and over time. This means the intended amount of reserve tokens needed for the wall protection may not be withdrawable, as the 4626 amount has increased beyond the approval amount.

Recommended mitigation

The ERC4626 asset approval should be closely tracked, possibly approved within the *swap()* function as well.

Team response

Not an issue with the current configuration (Using only sDAI).

TRST-M-2 Precision loss due to divide before multiply in several functions

- **Category:** Precision loss errors
- **Source:** UniswapV2PoolTokenPrice.sol, ERC4626Price.sol
- **Status:** Fixed

Description

Dividing before multiplying carries an inherent precision loss. Whereas each operation can loss accuracy by only one wei, when chaining operations this way the precision loss of the result is up to the amount multiplied by. That is what occurs in the code below, in UniswapV2PoolTokenPrice's *getPoolTokenPrice()*.

```
uint256 poolValue; // outputDecimals_  
{  
    uint256 priceMultiple = FixedPointMathLib.sqrt(  
        price0.mulDiv(price1, 10 ** outputDecimals_) * k  
    ); // sqrt(price * price) = outputDecimals_  
    uint256 two = 2 * 10 ** outputDecimals_;  
    poolValue = two.mulDiv(priceMultiple, poolSupply);  
}
```

Another instance is in ERC4626Price's *getPriceFromUnderlying()*.

```
// Get the number of underlying tokens per share  
// Scale: output decimals  
uint256 underlyingPerShare =  
asset.convertToAssets(assetScale).mulDiv(  
    outputScale,  
    assetScale  
);  
// Get the price of the underlying asset  
// We assume that getPrice() returns in outputDecimals  
// If the underlying price is not set, PRICE will revert  
uint256 underlyingPrice = _PRICE().getPrice(underlying);  
// Calculate the price of the asset  
// Scale: output decimals  
uint256 assetPrice =  
underlyingPrice.mulDiv(underlyingPerShare, outputScale);
```

If **assetScale > outputScale**, assets lose the remainder digits before being multiplied by the **underlyingPrice**.

Recommended mitigation

In both instances, aggregate all multiplications followed by all divisions.

Team response

Fixed.

Mitigation review

The two instances have been refactored successfully.

TRST-M-3 LBB0 can be manipulated through the BLVaultSupply module

- **Category:** MEV attacks
- **Source:** BLVaultSupply.sol
- **Status:** Acknowledged

Description

The BLVaultSupply module calculates the amount of collateralized OHM by checking the underlying vault balance and multiplying by the protocol's share:

```
function getCollateralizedOhm() external view override
returns (uint256) {
    // Prevent re-entrancy attacks
    VaultReentrancyLib.ensureNotInVaultContext(balVault);
    // Iterate through BLVaultManagers and total up the pool
    OHM share as the collateralized supply
    uint256 len = vaultManagers.length;
    uint256 total;
    for (uint256 i; i < len; ) {
        total += vaultManagers[i].getPoolOhmShare();
        unchecked {
            ++i;
        }
    }
    return total;
}
```

```
function getPoolOhmShare() public view override returns
(uint256) {
    // Cast addresses
    IVault vault = IVault(balancerData.vault);
    IBasePool pool = IBasePool(balancerData.liquidityPool);
    // Get pool total supply
    uint256 poolTotalSupply = pool.totalSupply();
    // Get token balances in pool
    (, uint256[] memory balances_, ) =
    vault.getPoolTokens(pool.getPoolId());
    // Balancer pool tokens are sorted alphabetically by
    token address. In the case of this
    // deployment, OHM is the first token in the pool.
    Therefore, the OHM balance is at index 0.
    if (poolTotalSupply == 0) return 0;
    else return (balances_[0] * totalLp) / poolTotalSupply;
}
```

Note that it takes the balances directly from the pool. The balancer vault balances can be easily manipulated by a sandwiching attack. Before the target pricing transaction, the attacker would inflate or deflate the ratio, and after the TX they would perform the reverse swap operation. This would cost some amount of swap fees. The SPPLY module results do not go through an aggregator like the PRICE submodules, therefore such a manipulation is guaranteed to work.

The collateralized OHM amount is used to calculate the liquid backing, which in turn calculates the LBBO parameter used as a fallback to the OHM target price as well as the RBS walls. Therefore, such a manipulation could have significant impact through abusing of the RBS system. Note that Operator *operate()* calls occur once every 8 hours, so it would take a while to undo the manipulated LBBO calculation.

Recommended mitigation

Use a separate price feed to determine the OHM value stored in the Vault managers and avoid basing it on reserve ratios.

Team response

BLV is being disabled, so we won't end up deploying BLV.

TRST-M-4 An attacker can deny protocol liveness by abusing deviation threshold

- **Category:** MEV attacks
- **Source:** BunniSupply.sol
- **Status:** Fixed

Description

The BunniSupply submodule uses the underlying Uni-V3 reserves to calculate the POL reserves. It makes sure the current reserves are not far off from the TWAP.

```
function _validateReserves(
    BunniKey memory key_,
    BunniLens lens_,
    uint16 twapMaxDeviationBps_,
    uint32 twapObservationWindow_
) internal view {
    uint256 reservesTokenRatio =
    BunniHelper.getReservesRatio(key_, lens_);
    uint256 twapTokenRatio =
    UniswapV3OracleHelper.getTWAPRatio(
        address(key_.pool),
        twapObservationWindow_
    );
    // Revert if the relative deviation is greater than the
    maximum
    if (
```

```
        // Not necessary to use `isDeviatingWithBpsCheck()`  
        as the checked is already performed in `addBunniToken`  
        Deviation.isDeviating(  
            reservesTokenRatio,  
            twapTokenRatio,  
            twapMaxDeviationBps_,  
            TWAP_MAX_DEVIATION_BASE  
        )  
    ) {  
        revert BunniSupply_PriceMismatch(  
            address(key_.pool),  
            twapTokenRatio,  
            reservesTokenRatio  
        );  
    }  
}
```

Note that the same code that limits the maximum manipulation amount will also trip any calls when a deviation is detected. Consider that every *operate()* call needs to resolve LBBO, which must fetch the reserve count. Therefore, an attacker can sandwich any *operate()* call with price manipulation of the v3 pool, making the protocol unresponsive. Of course, this would be at the cost of the manipulation swap fees, which depend on the specific parameters of the pool. At a minimum, this could be used to grief operators, making them lose gas fees for calling *operate()*. At the worst case, this could allow an exploitation of the current RBS parameters while the protocol does not update its key parameters.

Recommended mitigation

Refrain from using the current reserves as an indicator of price.

Team response

Fixed.

Mitigation review

The code was refactored and now only makes use of the current reserves, without checking for deviations.

TRST-M-5 The harvest reward calculation will always yield zero reward, disincentivizing harvests

- **Category:** Logical flaws
- **Source:** BunniManager.sol
- **Status:** Fixed

Description

The BunniManager calculates harvester reward below:

```
function getCurrentHarvestReward() public view override
bunniHubSet returns (uint256 reward) {
    // 0 if enough time has not elapsed
    if (lastHarvest + harvestFrequency < block.timestamp)
    return 0;
```

Note that when `block.timestamp` is below `lastHarvest + harvestFrequency`, it is too early to call `harvest()`:

```
uint48 minHarvest = lastHarvest + harvestFrequency;
if (minHarvest > block.timestamp) revert
BunniManager_HarvestTooEarly(minHarvest);
```

The condition in the line above should be flipped, allowing positive harvest rewards when it is time to collect.

Recommended mitigation

Flip the condition check in `getCurrentHarvestReward()`.

Team response

Fixed.

Mitigation review

The suggested fix was applied.

TRST-M-6 Bunni yield can be permanently stuck and uncollectable by the BunniManager

- **Category:** Logical flaws
- **Source:** BunniManager.sol, BunniHub.sol
- **Status:** Fixed

Description

Note: this finding has in parallel been reporting to the Bunni team as it affects their codebase, as well as the OlympusDAO code.

Bunni wraps Uniswap-v3 positions and allows pooling of assets in different concentrated liquidity ranges. Users can either join an existing Bunni range or create their own.

Lifecycle looks like:

- `deposit()` - inserts tokens and receive shares
- `withdraw()` - burns shares and receive equivalent liquidity in pool tokens
- `compound()` - collects accumulated uni-v3 fees and deposits them in the same position, increasing the value of user's shares as total liquidity increased.

Obviously, if a user deposits and withdraws without compounding, they won't receive yield. If they are the only share holder, that yield is erased, while if they aren't, it will be split between the other shareholders upon the next compound. Thus *compound()* must be used to receive yield.

There is significant oversight in the Bunni design stemming from the way *compound()* is implemented, relevant code is below:

```
// trigger an update of the position fees owed snapshots if
it has any liquidity
key.pool.burn(key.tickLower, key.tickUpper, 0);
(, , , uint128 cachedFeesOwed0, uint128 cachedFeesOwed1) =
key.pool.positions(
    keccak256(abi.encodePacked(address(this), key.tickLower,
key.tickUpper))
);
/// -----
---
/// amount0, amount1 are multi-purposed, see comments below
/// -----
---
amount0 = cachedFeesOwed0;
amount1 = cachedFeesOwed1;
/// -----
---
/// amount0, amount1 now store the updated amounts of fee
owed
/// -----
---
// the fee is likely not balanced (i.e. tokens will be left
over after adding liquidity)
// so here we compute which token to fully claim and which
token to partially claim
// so that we only claim the amounts we need
{
    (uint160 sqrtRatioX96, , , , , ) = key.pool.slot0();
    uint160 sqrtRatioAX96 =
TickMath.getSqrtRatioAtTick(key.tickLower);
    uint160 sqrtRatioBX96 =
TickMath.getSqrtRatioAtTick(key.tickUpper);
    // compute the maximum liquidity addable using the
accrued fees
    uint128 maxAddLiquidity =
LiquidityAmounts.getLiquidityForAmounts(
        sqrtRatioX96,
        sqrtRatioAX96,
        sqrtRatioBX96,
        amount0,
        amount1
    );
    // compute the token amounts corresponding to the max
addable liquidity
    (amount0, amount1) =
```

```

LiquidityAmounts.getAmountsForLiquidity(
    sqrtRatioX96,
    sqrtRatioAX96,
    sqrtRatioBX96,
    maxAddLiquidity
);
}
/// -----
---
/// amount0, amount1 now store the amount of fees to claim
/// -----
---
// the actual amounts collected are returned
// tokens are transferred to address(this)
(amount0, amount1) = key.pool.collect(
    address(this),
    key.tickLower,
    key.tickUpper,
    uint128(amount0),
    uint128(amount1)
);

```

It checks the total fees owed for the position, but only collects the optimal amount for adding liquidity, using LiquidityAmounts calculations. The rest is left as owed fees. If the current price is in position range, some ratio will be collected, otherwise, it will collect only one of the tokens and deposit it.

There are no additional access points to the fees owed of the position. This means that any yield generated that is currently not depositable as liquidity cannot be accessed by the user.

Suppose user opened their own solo ETH:USDC position for \$1500-\$1800 range. They've gathered a large amount of fees, and now ETH price moved to \$1801. At this point, the range is composed of only USDC, so the ETH fees cannot be accessed in any way.

In the context of OlympusDAO, the DAO operates a separate, permissioned BunniHub which only holds BunniManager's positions. Therefore the stuck yield is 100% at the expense of the DAO, which holds the entire Bunni position.

Recommended mitigation

Add a permissioned function in the BunniHub contract, which allows withdrawal of fees owed. This could be used as a last resort when fees are unclaimable.

Team response

Fixed.

Mitigation review

The *compound()* function now sends any non-compounded amount to the owner. This fixes the issue of possibly stuck tokens, however it should be noted that this may significantly impact the yield for compounding in Bunni positions. Consider that tokens often fluctuate

between different prices, so latter compounds are likely (but not guaranteed) to make use of the fees owed.

TRST-M-7 The protocol could face losses due to arbitrage through incorrect accounting of the Bunni holdings

- **Category:** Time-sensitivity issues
- **Source:** BunniManager.sol
- **Status:** Acknowledged

Description

The BunniManager's lifecycle is as follows:

- Register the PoolToken through *registerPool()* or *deployPoolToken()*
- Insert liquidity from the treasury through *deposit()*
- Call *activatePoolToken()* to add the token to the PRICE, TRSRY, SPPLY module accounting
- To finalize, *withdraw()* is used to collect back the underlying and send to the treasury.
- Then, *deactivatePoolToken()* would unregister from the modules

Note that *activatePoolToken()* must be called after deposits because it verifies there's liquidity.

```
// Check that the position has liquidity
if (
    !UniswapV3Positions.positionHasLiquidity(
        key.pool,
        key.tickLower,
        key.tickUpper,
        address(bunniHub)
    )
) revert BunniManager_PoolHasNoLiquidity(pool_);
```

The issue with the lifecycle is that between the time of deposit and time of activation, the protocol is blind to the asset pair held in the BunniManager, whilst beforehand they were accounted for in the treasury. This creates a theoretical fluctuation in the LBBO calculation, until they are accounted for again after the *activate()* call. If the heart beat is triggered at this time, the target OHM price could be wrong leading to arbitrage opportunities through the RBS stabilization schemes.

Recommended mitigation

The BunniManager should be restructured to ensure transfer of holdings from the treasury to the manager which is instantly visible to the Appraiser.

Team response

Won't fix. Activation scripts do this in a single transaction

TRST-M-8 The submodule observation array could be corrupted leading to wrong readings

- **Category:** Typo issues
- **Source:** OlympusSupply.sol
- **Status:** Fixed

Description

OlympusSupply manages a list of submodules which register for observations. Below it unregisters some submodule:

```
function unregisterFromObservations(SubKeycode subKeycode_) external override
permissioned {
    // Check if the submodule is already registered for observations
    bool registered;
    for (uint256 i; i < submodulesForObservationCount; ) {
        if (fromSubKeycode(submodulesForObservation[i]) ==
fromSubKeycode(subKeycode_)) {
            registered = true;
            break;
        }
        unchecked {
            ++i;
        }
    }
    // If it is not registered, revert
    if (!registered) revert SPPLY_InvalidParams();
    // Remove the submodule from the list of submodules for observation
    for (uint256 i; i < submodulesForObservationCount; ) {
        if (fromSubKeycode(submodulesForObservation[i]) ==
fromSubKeycode(subKeycode_)) {
            submodulesForObservation[i] = submodulesForObservation[submodules.length -
1];
            submodulesForObservation.pop();
            break;
        }
        unchecked {
            ++i;
        }
    }
    unchecked {
        --submodulesForObservationCount;
    }
}
```

The second loop identifies which index needs to be removed. It intends to overwrite it with the last element in **submodulesForObservation** and pop the array, however there is a typo-error so that it uses the **submodules.length**. That array includes all submodules, so the line would in fact potentially schedule two identical observations in the array, whilst removing the last observation. This would cause the moving average calculations to be wrong at the next interval.

Recommended mitigation

Use **submodulesForObservation.length** instead in the affected line.

Team response

Fixed.

Mitigation review

The issue was fixed as suggested.

TRST-M-9 The LBBO value will be incorrect after withdrawing from BunniSupply

- **Category:** Accounting issues
- **Source:** BunniSupply.sol
- **Status:** Acknowledged

Description

The Bunni submodule uses a moving average of the underlying reserves to calculate the amount of reserves it represents, which is used to calculate the LBBO (Liquid backing per backed Ohm). Note that the BunniManager can withdraw a position, sending the underlying tokens to the Treasury. The combination of a MA and an instant reading of the treasury reserves represents an accounting issue, whereby the reserves would be double-counted after a withdrawal (It would take a series of observations until the MA is wiped off). Similarly, when depositing into the BunniSupply, there would be undercounting of those reserves.

A deviation from the real reserves amount may be exploited by an attacker as it opens an arbitrage opportunity whenever the LBBO is used to set the OHM target price.

Recommended mitigation

To maintain accurate accounting, it is probably required to track additional state and take it into consideration when mixing between instant and MA value feeds.

Team Response

Acknowledged.

TRST-M-10 LBBO can be manipulated over time through the BunniSupply which uses current reserves

- **Category:** Logical flaws
- **Source:** BunniSupply.sol
- **Status:** Acknowledged

Description

The reserve values returned by BunniSupply are stretched over time by a moving average. However, note that at any particular *storeObservations()* interval, an attacker can manipulate the reserves by sandwiching the transaction. They could inflate one half of the holding pair by buying it before the *storeObservations()* and sell it afterwards. Over time, the LBBO could quite significantly diverge from its true value, introducing arbitrage opportunities against Ohm.

Recommended mitigation

There is probably no easy solution to the issue without additional refactoring.

Team response

Acknowledged

TRST-M-11 LBBO can be manipulated over time through the BunniPrice which uses current reserves

- **Category:** Logical flaws
- **Source:** BunniPrice.sol
- **Status:** Acknowledged

Description

The BunniManager registers a pricing strategy for each PoolToken in `_addPoolTokenToPRICE()`. In PR #354, the strategy is averaging the moving average and the instantaneous price of the token, calculated through current reserves. Therefore, the instantaneous price of the BunniManager can be arbitrarily manipulated by a resourced attacker. The second order effect, manipulating the LBBO, would occur over time as it is done through a moving average.

Recommended mitigation

There is probably no easy solution to the issue without additional refactoring.

Team response

Acknowledged

Low severity findings

TRST-L-1 In the Operator the bond pricing would be incorrect if price has 36 digits

- **Category:** Casting flaws
- **Source:** Operator.sol
- **Status:** Fixed

Description

In `_activate()`, the `oracleScale` is determined and later used to adjust the price.

```
// Calculate oracle scale and bond scale with scale
adjustment and format prices for bond market
oracleScale = 10 ** uint8(int8(_oracleDecimals) -
priceDecimals);
```

The `priceDecimals` is calculated using the function below:

```
function _getPriceDecimals(uint256 price_) internal view
returns (int8) {
    int8 decimals;
    while (price_ >= 10) {
        price_ = price_ / 10;
        decimals++;
    }
    // Subtract the stated decimals from the calculated
    decimals to get the relative price decimals.
    // Required to do it this way vs. normalizing at the
    beginning since price decimals can be negative.
    return decimals - int8(_oracleDecimals);
}
```

Note that the number inside `uint8()` wrapping could be negative, when there are over 36 digits in the price feed. When that occurs, the casting will make the negative number in the bracket become extremely large, due to how two's complement operates.

Recommended mitigation

Verify that the number is not negative before storing it in `oracleScale`.

Team response

Fixed.

Mitigation review

Overflow check has been inserted successfully.

TRST-L-2 Property getters would return incorrect data when max age is the current timestamp

- **Category:** Logical flaws
- **Source:** OlympusSupply.sol
- **Status:** Fixed

Description

Many property getters are implemented in the Olympus contracts, for example:

```
function getMetric(Metric metric_, uint48 maxAge_) external
view override returns (uint256) {
    // Get the cached value of the metric
    (uint256 value, uint48 timestamp) = getMetric(metric_,
Variant.LAST);
    // Try to use the last value, must be no older than
maxAge_
    if (timestamp >= uint48(block.timestamp) - maxAge_)
return value;
    // If the last value is not on the current timestamp,
calculate the current value
    (value, ) = getMetric(metric_, Variant.CURRENT);
    return value;
}
```

There is a well documented behavior, which states that Variant.LAST can return (0,0) if it was never set.

```
/// @inheritdoc SPPLYv1
/// @dev If the `Variant.LAST` variant is requested
and it has not yet been stored, then (0, 0) will be
returned.
///
/// @dev Will revert if:
/// @dev - The value for `metric_` is invalid
/// @dev - The value for `variant_` is invalid
function getMetric(
    Metric metric_,
    Variant variant_
) public view override returns (uint256, uint48) {
```

However it is a somewhat dangerous and undocumented behavior that this could be returned from the function accepting **maxAge**. In case the age is block.timestamp, the value would be accepted. Semantically it would be unexpected for the caller of *getMetric()* to get a value never set, and there may be ways for attackers to control **maxAge** in integrations with the call.

Recommended mitigation

When timestamp is zero in *getMetric()* and similar functions, revert the call.

Team response

Fixed.

Mitigation review

All functions with **maxAge** parameter will revert when it is higher or equal to the current timestamp, therefore it will never accept a cached value with timestamp of zero.

TRST-L-3 `isDeviating()` suffers from rounding issue, erring by up to 1bps

- **Category:** Precision loss errors
- **Source:** Deviation.sol, PoolLibrary.sol
- **Status:** Fixed

Description

The `_isDeviating()` function should detect deviation that is above the specified threshold.

```

/// @notice Checks if the deviation between
two values is greater than the given deviation
/// @dev This function will revert if:
/// - `benchmark_` is zero
///
/// @param diff_ The difference between the two
values
/// @param benchmark_ The reference value to check
against
/// @param deviationBps_ The deviation in basis points
(e.g. 100 = 1%)
/// @param deviationMax_ The maximum deviation in basis
points (e.g. 10000 = 100%)
/// @return True if the deviation is greater
than the given deviation, false otherwise
function _isDeviating(
    uint256 diff_,
    uint256 benchmark_,
    uint256 deviationBps_,
    uint256 deviationMax_
) internal pure returns (bool) {
    return (diff_ * deviationMax_) / benchmark_ >
deviationBps_;
}

```

Suppose **diff** is **19999**, **benchmark** is **100000000**, **deviationBps_** is **1**. Then the output would be **1 > 1 = False**. But the actual deviation is **1.9999** bps while threshold is **1**, so the function would pass despite deviation being off by almost 100%.

A similar issue manifests in PoolLibrary `getAmountMin()`:

```

function getAmountMin(uint256 amount_, uint16 slippageBps_)
internal pure returns (uint256) {
    // Check bounds
    if (slippageBps_ > SLIPPAGE_SCALE) revert
InvalidSlippage(slippageBps_, SLIPPAGE_SCALE);
}

```

```
return (amount_ * uint256(SLIPPAGE_SCALE -
slippageBps_)) / uint256(SLIPPAGE_SCALE);
}
```

The function should return no less than the stated BPS of slippage, however due to rounding down in the division stage the result may be above this slippage.

Recommended mitigation

Use *mulDivUp()* when performing calculations such as the above.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-4 The ERC4626 pricer does not support tokens that comply with the spec and use a popular token mechanic

- **Category:** Compatibility issues
- **Source:** ERC4626Price.sol
- **Status:** Acknowledged

Description

The ERC4626 pricer submodule performs the check below on the target ERC4626 contract.

```
uint8 assetDecimals = asset.decimals();
uint8 underlyingDecimals = ERC20(underlying).decimals();
// This shouldn't be possible, but we check anyway
if (assetDecimals != underlyingDecimals) {
    revert ERC4626_AssetDecimalsMismatch(assetDecimals,
underlyingDecimals);
}
```

It would revert when the *decimals()* are different between the vault and the base asset. However, the Open Zeppelin and solmate implementations both have support out-of-the-box for virtual shares, which operate by incrementing the decimals of the base asset. For the sake of composability, a ERC4626 pricer should work with popular token mechanics.

Recommended mitigation

It is not required to perform the check above.

Team response

Only planning to use sDAI at the moment. No plans to use an ERC4626 token with differing decimals.

TRST-L-5 BunniManager does not assign categories for all groups when adding pool tokens.

- **Category:** Logical flaws
- **Source:** BunniManager.sol
- **Status:** Fixed

Description

The BunniManager registers a pool token as an asset in the code below:

```
function _addPoolTokenToTRSRY(address pool_, address
poolToken_) internal {
    TRSRYv1_1.Asset memory assetData =
    TRSRY.getAssetData(poolToken_);
    // Revert if the asset exists and the location is
    already registered (which would indicate an inconsistent
    activation state)
    if (assetData.approved == true) {
        bool locationExists = false;
        for (uint256 i = 0; i < assetData.locations.length;
i++) {
            if (assetData.locations[i] == address(TRSRY)) {
                locationExists = true;
                break;
            }
        }
        if (locationExists) revert
BunniManager_TokenActivated(pool_, toKeycode("TRSRY"));
    } else {
        // Add the asset to TRSRY
        TRSRY.addAsset(poolToken_, new address[](0));
    }
    // Add TRSRY as a location
    TRSRY.addAssetLocation(poolToken_, address(TRSRY));
    // Categorize the asset
    TRSRY.categorize(poolToken_,
toTreasuryCategory("protocol-owned-liquidity"));
}
```

Note that it does not register in the two categories: **market-sensitivity** and **liquidity-preference**. When not registering there may be wrong assumptions in the code as generally an asset should belong to exactly one category in each group. For example, the code below assumes any asset not in the **illiquid** category is liquid.

```
function _liquidBacking() internal view returns (uint256) {
    // Get total backing
    uint256 backing = _backing();
}
```

```
// Get value of assets categorized as illiquid
(uint256 illiquidValue, ) =
_categoryValue(toTreasuryCategory("illiquid"));
// Subtract illiquid value from total backing
return backing - illiquidValue;
}
```

Recommended mitigation

Categorize across all category groups.

Team response

BunniManager is now configured as **liquid** and **volatile**.

TRST-L-6 Liquid backing calculation would be wrong if complex assets including OHM would be marked illiquid

- **Category:** Validation issues
- **Source:** Appraiser.sol
- **Status:** Fixed

Description

The liquid backing is defined below:

```
function _liquidBacking() internal view returns (uint256) {
    // Get total backing
    uint256 backing = _backing();
    // Get value of assets categorized as illiquid
    (uint256 illiquidValue, ) =
_categoryValue(toTreasuryCategory("illiquid"));
    // Subtract illiquid value from total backing
    return backing - illiquidValue;
}
```

The `_backing()` calculation accumulates all the non-OHM reserves in the treasury and in the supply modules. `liquidBacking()` reduces that amount by the value of all assets in the **illiquid** category. Note that this calculation does not verify the assets accumulated are not OHM.

```
function _categoryValue(TreasuryCategory category_) internal
view returns (uint256, uint48) {
    // Get the assets in the category
    address[] memory assets =
TRSTRY.getAssetsByCategory(category_);
    // Get the value of each asset in the category
    uint256 value;
    uint256 len = assets.length;
    for (uint256 i; i < len; ) {
```

```

        (uint256 assetValue, ) = _assetValue(assets[i]);
        value += assetValue;
        unchecked {
            ++i;
        }
    }
    return (value, uint48(block.timestamp));
}

```

```

function _assetValue(address asset_) internal view returns
(uint256, uint48) {
    // Get current asset price, should be in price decimals
    configured on PRICE
    (uint256 price, ) = PRICE.getPrice(asset_,
PRICEv2.Variant.CURRENT);
    // Get current asset balance, should be in the decimals
    of the asset
    (uint256 balance, ) = TRSRY.getAssetBalance(asset_,
TRSRYv1_1.Variant.CURRENT);
    // Calculate the value of the protocols holdings of the
    asset
    uint256 value = (price * balance) / (10 **
ERC20(asset_).decimals());
    return (value, uint48(block.timestamp));
}

```

Therefore, if assets that are composed of OHM/gOHM are registered to the treasury as illiquid, they will reduce *liquidBacking()* by their notional value while the positive side is never registered in *backing()*, causing undercounting of assets.

Recommended mitigation

Consider refactoring the logic of *liquidBacking()* to reduce the risk of miscounting.

Team response

Fixed.

Mitigation review

Fixed by improving the logic of *liquidBacking()*.

TRST-L-7 *thirtyDayOhmVolatility()* does not use Bessel's correction

- **Category:** Mathematical flaws
- **Source:** Appraiser.sol
- **Status:** Acknowledged

Description

_thirtyDayOhmVolatility() uses 89 samples to calculate the 8-hour volatility, which is then annualized. To do so, the data is first used to calculate the mean value by dividing the sum by

the number of samples, i.e. by 89. In order to get the variance, the sum of squared residuals is calculated, which is then divided by the number of samples (89). Ultimately, the square root is taken to get the standard deviation.

This estimator for the volatility is biased because the data is used to get the mean, which reduces the degrees of freedom in the residuals vector by 1. An unbiased estimator of the variance can be achieved by applying Bessel's correction, i.e. dividing the sum of squared residuals by 88 instead of 89. Note that this results in a completely unbiased estimator for the variance, but not for the standard deviation (i.e. the volatility). It will still be biased (and there is no general unbiased estimator of it), but the bias will be reduced by applying the correction.

Recommended mitigation

Divide by 88 instead in the formula.

Team response

Function isn't used.

TRST-L-8 Appraiser miscalculates annual volatility by 0.1%

- **Category:** Precision loss issues
- **Source:** Appraiser.sol
- **Status:** Fixed

Description

The `_thirtyDayOhmVolatility()` converts standard deviation to volatility in the code below:

```
// Calculate and return annual volatility
return stdDev * 33; // annual std dev = period std dev *
sqrt(periods per year), in this case there are 365 * 3 =
1095 periods per year. sqrt(1095) = 33.097...
```

There is an unnecessary precision loss due to multiplying by 33 instead of 33.090784215548595. The numbers are already stored at 1e18 precision so the improved accuracy is essentially free.

Recommended mitigation

Multiply by `sqrt(1095)` in 1e18 decimals.

Team response

Fixed.

Mitigation review

Issue has been addressed. Consider replacing with a constant to reduce gas costs.

TRST-L-9 OlympusPriceV2 does not mark feed as unsuccessful if it returns a price of zero

- **Category:** Validation issues
- **Source:** OlympusPrice.v2.sol
- **Status:** Fixed

Description

In `_getCurrentPrice()`, the pricer loops over all feeds and marks if any did not succeed.

```
(bool success_, bytes memory data_) =  
address(_getSubmoduleIfInstalled(feeds[i].target))  
    .staticcall(  
        abi.encodeWithSelector(feeds[i].selector, asset_,  
_decimals, feeds[i].params)  
    );  
// Store price if successful, otherwise leave as zero  
// Idea is that if you have several price calls and just  
// one fails, it'll DOS the contract with this revert.  
// We handle faulty feeds in the strategy contract.  
if (success_) {  
    prices[i] = abi.decode(data_, (uint256));  
} else {  
    successAllFeeds = false;  
}
```

Note that if a feed returns a price of zero, it is most certainly not operational, and should be marked as not successful. In the worst case, a faulty price feed will not be detected when adding an asset and checking for success:

```
// Validate configuration  
(, , bool successAllFeeds) = _getCurrentPrice(asset_, true);  
if (!successAllFeeds) revert  
PRICE_PriceFeedCallFailed(asset_);
```

Recommended mitigation

Check the return value for zero.

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-10 addBunniToken() allows configuration of insufficient TWAP period

- **Category:** Configuration issues
- **Source:** BunniSupply.sol
- **Status:** Fixed

Description

The minimum TWAP period supported when adding a BunniToken is 19 seconds.

```
if (twapObservationWindow_ <
    UniswapV3OracleHelper.TWAP_MIN_OBSERVATION_WINDOW)
    revert BunniSupply_Params_InvalidTwapObservationWindow(
        token_,
        UniswapV3OracleHelper.TWAP_MIN_OBSERVATION_WINDOW,
        twapObservationWindow_
    );
```

```
uint32 internal constant TWAP_MIN_OBSERVATION_WINDOW = 19;
// seconds
```

19 seconds cannot be an acceptable TWAP period as it would be sufficient to manipulate a single block to control the TWAP.

Recommended mitigation

Consider using a minimum TWAP period of at least 30 minutes.

Team response

Fixed.

Mitigation review

The TWAP minimum period is now 10 minutes.

TRST-L-11 LBBO may be miscalculated due to the SiloSupply module

- **Category:** Logical flaws
- **Source:** SiloSupply.sol
- **Status:** Acknowledged

Description

The LBBO in Olympus serves as the ratio which OHM holders should be confident they are able to redeem OHM at, assuming the protocol-owned liquidity and treasury to never cash out OHM. To calculate the "backed OHM" part of LBBO, it takes the floating supply and reduces it by the collateralized OHM supply. This is generally safe because the latter is OHM that is backed by assets not visible to the treasury, but they are held in place to cover the OHM.

The issue is that the SiloSupply module interprets borrowed OHM as collateralized.

```
function getCollateralizedOhm() external view override
returns (uint256) {
    // Get OHM collateral token for this silo
    address ohmCollateralToken =
silo.assetStorage(ohm).collateralToken;
    // Get amount of OHM supplied to market by lending
facility
    uint256 totalDeposits =
lens.totalDepositsWithInterest(silo, ohm);
    uint256 supplied =
lens.balanceOfUnderlying(totalDeposits, ohmCollateralToken,
amo);
    // Get amount of OHM borrowed from silo
    uint256 borrowed =
lens.totalBorrowAmountWithInterest(silo, ohm);
    // If supplied > borrowed, then borrowed is
collateralized supply
    // Otherwise, supplied is collateralized supply
    return supplied > borrowed ? borrowed : supplied;
}
```

Suppose 1M ohm is deposited to the Silo and subsequently borrowed by an attacker. This would not change the backed amount, because the borrowable amount which is reduced in *_floatingSupply()* calculation becomes collateralized supply, reduced in *backedSupply()*. However logically it is an amount that needs to be backed, because it could be dumped on the market for the LBBO value. It is collateralized in the sense that user has put deposited money to cover the borrowing in the Silo, however that unaccounted amount is not the protocol's secured amount, like in the BLVaultSupply contract.

In the worst case, a large Silo deposit could be exploited by a user to borrow and redeem OHM tokens for their inflated LBBO value, eventually making last redeemers not able to get the same amount.

Recommended mitigation

getCollateralizedOhm() should presumably return zero.

Team response

We were aware of this potential dynamic. It requires newly minted OHM from the AMO to be deposited into Silo, which is an admin activity. Nonetheless, it could be an issue in the future. The planned mitigation is to limit the amount of OHM allowed to be minted in these markets relative to the backed supply to avoid major risk.

TRST-L-12 OlympusPriceV2 *getCurrentPrice()* may use a cached price

- **Category:** Logical flaws
- **Source:** OlympusPrice.v2.sol
- **Status:** Fixed

Description

The `getCurrentPrice()` function of `OlympusPriceV2` should presumably never use a cached price. However, when it queries some price feeds they will accept a cached value from the last block:

- `BunniPrice`'s `getBunniTokenPrice()` calls `_getTotalValue()`, which calls `_PRICE().getPrice(token0)`
- `ERC4626Price` `getPriceFromUnderlying()` calls `_PRICE().getPrice(underlying)`

Note that all other pricers use the more secure variant, `PRICE.getPrice(X, PRICEv2.Variant.CURRENT)`.

Recommended mitigation

Consider refactoring the above instances to use the current token pricing.

Team response

Fixed.

Mitigation review

Issue was fixed as suggested.

TRST-L-13 Reserve calculation may count GOHM as part of backing value

- **Category:** Accounting issues
- **Source:** `Appraiser.sol`
- **Status:** Fixed

Description

In order to calculate backing value, `Appraiser` loops over the POL reserves and increases by any non-OHM reserve:

```
// Get the POL reserves from the SPPLY module
SPPLYv1.Reserves[] memory reserves =
SPPLY.getReservesByCategory(
    toSupplyCategory("protocol-owned-liquidity")
);
len = reserves.length;
for (uint256 i; i < len; ) {
    uint256 tokens = reserves[i].tokens.length;
    for (uint256 j; j < tokens; ) {
        if (reserves[i].tokens[j] != ohm) {
            // Get current asset price
            (uint256 price, ) = PRICE.getPrice(
                reserves[i].tokens[j],
                PRICEv2.Variant.CURRENT
            );
        }
    }
}
```

```

        // Calculate current asset valuation
        value +=
            (price * reserves[i].balances[j]) /
            (10 **
ERC20(reserves[i].tokens[j]).decimals());
    }
    unchecked {
        ++j;
    }
}
unchecked {
    ++i;
}
}
return value;

```

If there are Reserves that include gOHM, these will be counted as part of the backing incorrectly.

Recommended mitigation

Make sure gOHM is also not included in the calculation.

Team response

Fixed.

Mitigation review

During calculation of backing reserves and supply amounts, gOHM is now ignored. Note that the check below is unnecessary, because `_assetValue()` will safely ignore `ohm` and `gOhm` assets.

```

if (assets[i] != ohm && assets[i] != gohm) {
    (uint256 assetValue, ) = _assetValue(assets[i],
true);
    if (!_inArray(assets[i], polAssets)) {
        value += assetValue;
    }
}

```

```

function _assetValue(address asset_, bool excludeOhm_)
internal view returns (uint256, uint48) {
    if (excludeOhm_ == true && (asset_ == ohm || asset_ ==
gohm)) {
        return (0, uint48(block.timestamp));
    }
}

```

TRST-L-14 Tight coupling of modules and categories

- **Category:** Architecture issues
- **Source:** OlympusSupply.sol
- **Status:** Acknowledged

Description

The current design forces modules to implement each of the methods any of the categories use, otherwise the code below would revert.

```
for (uint256 i; i < len; ) {
    address submodule =
address(_getSubmoduleIfInstalled(submodules[i]));
    (bool success, bytes memory returnData) =
submodule.staticcall(
    abi.encodeWithSelector(data.submoduleMetricSelector)
);
    // Ensure call was successful
    if (!success)
        revert SPPLY_SubmoduleFailed(address(submodule),
data.submoduleMetricSelector);
}
```

This limits the versatility of the contract significantly. New categories that would like to make use of a different selector can never be used as they would cause reverts on older modules. Also, a lot of boilerplate code is necessary for every module.

Recommended mitigation

Relax restrictions around calls to custom selectors. Assume no implementation of a method is the same as zero-value.

Team response

Low priority – won't fix for now.

TRST-L-15 The wall may be depleted sooner than intended in Operator

- **Category:** Logical flaws
- **Source:** Operator.sol
- **Status:** Acknowledged

Description

The wall's full capacity is calculated in the function below:

```
/// @inheritdoc IOperator
function fullCapacity(bool high_) public view override
returns (uint256) {
    uint256 reservesInTreasury =
_wrappedReserve.previewRedeem(
    TRSRY.getReserveBalance(_wrappedReserve)
) + TRSRY.getReserveBalance(_reserve);
    uint256 capacity = (reservesInTreasury *
```

```
_config.reserveFactor) / ONE_HUNDRED_PERCENT;
    if (high_) {
        capacity =
            (capacity.mulDiv(
                10 ** _ohmDecimals * 10 ** _oracleDecimals,
                10 ** _reserveDecimals * RANGE.price(true,
true)
            ) * (ONE_HUNDRED_PERCENT + RANGE.spread(true,
true) + RANGE.spread(false, true))) /
            ONE_HUNDRED_PERCENT;
    }
    return capacity;
}
```

It sums the reserve amount of the unwrapped equivalent of wrapped reserves and the plain reserves. The issue is that the *swap()* function assumes the capacity is all available in wrapped form:

```
// Calculate amount of wrappedReserve equivalent to
amountOut
// and withdraw wrapped reserves from TRSRY
TRSRY.withdrawReserves(
    address(this),
    _wrappedReserve,
    _wrappedReserve.previewWithdraw(amountOut)
);
// Unwrap reserves and transfer to sender
_wrappedReserve.withdraw(amountOut, msg.sender,
address(this));
```

This could cause the wall to not satisfy swap requests although there are enough plain reserves to satisfy it.

Recommended mitigation

In *swap()*, verify there are sufficient wrapped reserves when using the *withdrawReserves()* function. Otherwise, withdraw from the plain reserves.

Team response

In the past, we expected the Treasury management team to ensure there was enough of the required token in the treasury (whether wrapped or not) so that the wall functions correctly. Given the way the wrapped tokens are managed and the bytecode limit for the Operator, we don't think this needs to be added.

TRST-L-16 Leaked approval in the BunniManager

- **Category:** Logical flaws
- **Source:** BunniManager.sol
- **Status:** Fixed

Description

When depositing into the BunniManager, it approves the **token0**, **token1** amounts to the BunniHub, then calls *deposit()*.

```
ERC20(token0Address).approve(address(bunniHub),
token0Amount);
ERC20(token1Address).approve(address(bunniHub),
token1Amount);
}
// Construct the parameters
IBunniHub.DepositParams memory params =
IBunniHub.DepositParams({
    key: key,
    amount0Desired: token0Amount,
    amount1Desired: token1Amount,
    amount0Min:
UniswapV3PoolLibrary.getAmountMin(token0Amount,
slippageBps_),
    amount1Min:
UniswapV3PoolLibrary.getAmountMin(token1Amount,
slippageBps_),
    deadline: block.timestamp, // Ensures that the action
be executed in this block or reverted
    recipient: address(TRSRY) // Transfers directly into
TRSRY
```

The issue is that *deposit()* may not consume the entire approval, since the **amount0Min**, **amount1Min** parameters allow some slippage. In case the approval is not used, it will be leaked. There is no clear way to abuse it, but note that tokens which revert when changing approval from non-zero to a non-zero amount, like USDT, will now revert.

Recommended mitigation

Reset approvals to zero at the end of the affected function.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-17 The Config contracts are not able to use all the necessary Module actions

- **Category:** Logical flaws
- **Source:** TreasuryConfig.sol, SupplyConfig.sol
- **Status:** Fixed

Description

The TreasuryConfig is delegated authority to configure the treasury. It requires access to the following functions:

```
function requestPermissions() external view override returns
(Permissions[] memory requests) {
    Keycode TRSRY_KEYCODE = TRSRY.KEYCODE();
    requests = new Permissions[](6);
    requests[0] = Permissions(TRSRY_KEYCODE,
    TRSRY.addAsset.selector);
    requests[1] = Permissions(TRSRY_KEYCODE,
    TRSRY.addAssetLocation.selector);
    requests[2] = Permissions(TRSRY_KEYCODE,
    TRSRY.removeAssetLocation.selector);
    requests[3] = Permissions(TRSRY_KEYCODE,
    TRSRY.addCategoryGroup.selector);
    requests[4] = Permissions(TRSRY_KEYCODE,
    TRSRY.addCategory.selector);
    requests[5] = Permissions(TRSRY_KEYCODE,
    TRSRY.categorize.selector);
}
```

In fact, there are several functions which should also be callable:

- *removeAsset()*
- *removeCategoryGroup()*
- *removeCategory()*
- *uncategorize()*
- *storeBalance()*

Similarly, SupplyConfig requests the following permissions:

```
/// @inheritdoc Policy
function requestPermissions() external view override returns
(Permissions[] memory requests) {
    Keycode SPPLY_KEYCODE = toKeycode("SPPLY");
    requests = new Permissions[](6);
    // SPPLY Permissions
    requests[0] = Permissions(SPPLY_KEYCODE,
    SPPLY.addCategory.selector);
    requests[1] = Permissions(SPPLY_KEYCODE,
    SPPLY.removeCategory.selector);
    requests[2] = Permissions(SPPLY_KEYCODE,
    SPPLY.categorize.selector);
    requests[3] = Permissions(SPPLY_KEYCODE,
    SPPLY.installSubmodule.selector);
    requests[4] = Permissions(SPPLY_KEYCODE,
    SPPLY.upgradeSubmodule.selector);
    requests[5] = Permissions(SPPLY_KEYCODE,
    SPPLY.execOnSubmodule.selector);
}
```

It is missing:

- `storeCategorySupply()`
- `storeMetric()`

Recommended mitigation

Add wrapped functions for the functions listed and request permissions to use them.

Team response

Fixed.

Mitigation review

Some additional functions are exposed. The others are not open to `SupplyConfig` and `TreasuryConfig` by design.

TRST-L-18 Validation of heartbeat frequency is missing when distributor is updated

- **Category:** Validation issues
- **Source:** Heart.sol
- **Status:** Fixed

Description

PR #351 introduces a check that the heartbeat frequency is synced to the staking distributor epoch.

```
// Sync heartbeat with staking contract
(uint256 epochLength, , uint256 epochEnd, ) =
IStaking(distributor.staking()).epoch();
if (frequency() != epochLength) revert Heart_InvalidFrequency();
lastBeat = uint48(epochEnd - epochLength);
```

An issue occurs because the distributor can also be set again by an admin.

```
/// @inheritdoc IHeart
function setDistributor(address distributor_) external
onlyRole("heart_admin") {
    distributor = IDistributor(distributor_);
}
```

An admin could mistakenly set a distributor which isn't aligned to the existing heartbeat.

Recommended mitigation

When the distributor changes, validate the epoch length is the same as `frequency()`.

Team response

Fixed.

Mitigation review

Issue was addressed as suggested.

TRST-L-19 A metric that has been removed and added to tracking can be stale

- **Category:** Logical flaws
- **Source:** Appraiser.sol
- **Status:** Acknowledged

Description

PR #339 introduces tracking of metric observations through the Heart *beat()*. These can be removed and added by the **heart_admin** role. The function which updates with the last observation is Appraiser's *storeMetricObservation()*. It will fetch the latest price and update the cumulative counter as well as the observation array.

Suppose the following scenario:

- A metric is added to the Heart tracking, and registers observations
- It is removed from Heart tracking
- After some time, it is re-added

At this point, the next call to *storeMetricObservation()* would add the new result directly after the historical reading. It would cause the moving average calculation to be off, because it's using stale data.

Recommended mitigation

In *storeMetricObservation()*, validate that the time elapsed since the last update is not above a certain staleness threshold. If so, the previous data must be wiped through *updateMetricMovingAverage()* by the admin first.

Team response

Acknowledged, won't be fixed in this round.

TRST-L-20 Asset price can be updated twice per period

- **Category:** Logical flaws
- **Source:** OlympusPrice.v2.sol
- **Status:** Acknowledged

Description

PR #354 refactors price observations, so that the heart calls *storeObservations()* of the OlympusPrice. It loops over all assets that are tracking a moving average and update it. Additionally, *storePrice()* is a permissioned function which updates the price of a specific asset.

The design is error prone as at no point do either function check that the previous observation is not recent. If an admin calls *storePrice()*, the execution of that call is unpredictable and may be close to the next or previous beat. That would skew the moving average which may be exploitable.

Recommended mitigation

In *storePrice()*, if the previous result is significantly below the update frequency, skip the observation.

Team response

Acknowledged

TRST-L-21 If BunniSupply contains positions with gOHM, LBBO calculation will be wrong

- **Category:** Logical flaws
- **Source:** BunniSupply.sol
- **Status:** Acknowledged

Description

In BunniSupply, the code below determines the amount of Ohm reserves held, used by the *getProtocolOwnedLiquidityOhm()* API:

```
function _getOhmReserves (
    BunniKey memory key_,
    TokenMovingAverage memory tokenMovingAverage_
) internal view returns (uint256) {
    if (key_.pool.token0() == ohm) {
        return
            tokenMovingAverage_.token0CumulativeObservations
        /
            tokenMovingAverage_.numObservations;
    } else if (key_.pool.token1() == ohm) {
        return
            tokenMovingAverage_.token1CumulativeObservations
        /
            tokenMovingAverage_.numObservations;
    } else {
        return 0;
    }
}
```

Note that gOHM is semantically the same as OHM, and should count as Ohm for backing calculations. However the code does not check for one of the tokens being gOHM.

Recommended mitigation

Check if **token0** or **token1** are Ohm or gOhm.

Team response

Acknowledged

Additional recommendations

TRST-R-1 Docs improvements

- OlympusTreasury *addAsset()* - also reverts if a location does not already exist in the array
- SimplePriceFeedStrategy *_getMedianPrice()* - It supports 2-price arrays as well (optional)
- Oracle *getTWAPRatio()* - token0 / token1 names are misleading, they are base token and quote token respectively.
- BunniManager *_removePoolTokenFromTRSRY()* – does not actually remove categorization of the asset.
- BunniManager documentation of **pools** is incorrect – The **pools** members are not necessarily deployed by the contract, they could have been directly registered.

TRST-R-2 Safety validations

- *removeAsset()* - consider reverting if the asset is not found in the assets array.
- Use *mulDiv()* consistently when multiplying and dividing by large numbers, e.g. in *_backing()*

```
// Calculate current asset valuation
value +=
    (price * reserves[i].balances[j]) /
    (10 ** ERC20(reserves[i].tokens[j]).decimals());
```

- In BunniManager's *_addPoolTokenToTRSRY()*, consider checking the previous **poolToken** does not belong to another category before categorizing it.

TRST-R-3 Gas Optimizations

There are many instances (e.g. *getAssetsByCategory()*) when there are two loops, one to count the amount to allocate memory for, and another to copy into memory. This effectively doubles the amount of SLOADs being expended, which is very significant. Consider a different approach such as manual memory management in YUL.

TRST-R-4 Readability

- In OlympusSupply, use *.selector* syntax instead of hardcoded byte strings

```
_addCategory(toCategory("protocol-owned-treasury"), true,
0xb600c5e2, 0x00000000); // getProtocolOwnedTreasuryOhm()
```

- `_thirtyDayOhmVolatility()` name is misleading, as it calculates 365-day ohm volatility using 30 days of data.

TRST-R-5 Coding style

- In OlympusSupply `getCategoryByLocation()` consider reverting when category is not found, similarly to `getCategoryData()` or `getLocationsByCategory()`. This helps prevent zero-initialized memory issues.
- There are multiple structs called Asset in the codebase.
- Abstract functions should not be implemented as empty functions at the base contract. This pattern is error-prone as implementations may skip implementing them. Relevant for Submodule, Module and Policy contracts.

TRST-R-6 Price feed strategy

- The `getAveragePrice()`, `getMedianPrice()` and other variants will work with just one operational feed. However that behavior is probably undesirable when there is large redundancy in oracles (e.g. 1/10 is working). Consider adding a minimum-feeds variable to achieve higher reliability on feeds.

TRST-R-7 Decoupling decimal assumptions

- In `_liquidBackingPerBackedOhm()`, it divides by two variables in OHM decimals (9). It adjusts by multiplying by **priceScale**, which is deployed as 18 but doesn't *have* to be. Semantically it is coupling configuration assumptions unnecessarily.

TRST-R-8 Unused variables

- In `removeBunniToken()`, **bunniTokenIndex** is never used. Consider removing it, or better, verifying it is not **max** after the loop is complete.

TRST-R-9 Improve reliability of the protocol

The Heart `beat()` function checks a lower bound but not an upper bound for the time since last invocation. The moving average assumes `beat()` calls are 8-hours apart, so it would be best to:

- Correct for drifts in the *beat()* frequency
- Enforce an upper bound for *beat()* calls

TRST-R-10 Improve versatility of BrickedSupply contract

Currently, the contract only calculates bricked supply in the form of tokens stuck in the token contract. However there are certainly more bricked tokens in other “dead” addresses. It is recommended to supply a “dead addresses” list to improve the accuracy of bricked supply.

TRST-R-11 Improve resistance to manipulation of the LBBO

The following functions are permissionless in the Appraiser contract:

```
/// @inheritdoc IAppraiser
function storeAssetValue(address asset_) external override {
    (uint256 value, uint48 timestamp) =
    getAssetValue(asset_, Variant.CURRENT);
    assetValueCache[asset_] = Cache(value, timestamp);
}
/// @inheritdoc IAppraiser
function storeCategoryValue(TreasuryCategory category_)
external override {
    (uint256 value, uint48 timestamp) =
    getCategoryValue(category_, Variant.CURRENT);
    categoryValueCache[category_] = Cache(value, timestamp);
}
/// @inheritdoc IAppraiser
function storeMetric(Metric metric_) external override {
    (uint256 result, uint48 timestamp) = getMetric(metric_,
    Variant.CURRENT);
    metricCache[metric_] = Cache(result, timestamp);
}
```

Note that as they are permissionless, an attacker can call them inside a flash loan, storing a manipulated value which would later be used by a *getMetric()* call for example, to get the LBBO.

TRST-R-12 Improve emission of events

In the BunniManager the pool can be registered through either *registerPool()* or *deployTokenPool()*, which also registers it. However the **PoolTokenRegistered** event is only emitted in the first function.

TRST-R-13 Improve slippage check at BunniManager

When withdrawing from the BunniManager, the admin passes **slippageBPS**. This factor is applied on the amounts calculated from the ratio of the current reserves.

```
(uint256 amount0, uint256 amount1) =
UniswapV3Positions.getPositionAmounts(
    key.pool,
    key.tickLower,
    key.tickUpper,
    address(bunniHub)
);
// Adjust for proportion of total supply
uint256 totalSupply = existingToken.totalSupply();
amount0 = amount0.mulDiv(shares_, totalSupply);
amount1 = amount1.mulDiv(shares_, totalSupply);
amount0Min = UniswapV3PoolLibrary.getAmountMin(amount0,
slippageBps_);
amount1Min = UniswapV3PoolLibrary.getAmountMin(amount1,
slippageBps_);
```

In fact, it means the slippage check does not exist, because it is applied on the current reserve ratio. Attacker can sandwich the transaction to make the manager withdraw at a skewed ratio. To the best of our knowledge, there's no direct way to exploit it under the assumption the pool is already showing the true price, but it is still best practice to pass per-reserve slippages similar to Uniswap's *removeLiquidity()* parameters.

TRST-R-14 Potentially trimming issues in observation calculations

In PR #339, the Appraiser makes use of the ROLES module. However, it does not change the expected module version array.

```
bytes memory expected = abi.encode([2, 1, 1]);
if (PRICE_MAJOR != 2 || SPPLY_MAJOR != 1 || TRSRY_MAJOR != 1
|| ROLES_MAJOR != 1)
    revert Policy_WrongModuleVersion(expected);
```

Consider adding another **1** to the array.

TRST-R-15 Setting unchangeable variables as immutable

PR #339 introduces the **observationfrequency** variable in Appraiser. Since it is set in the constructor and cannot be reset, it is best to define as immutable for gas optimization and readability.

TRST-R-16 Add a staleness bit for moving average data in Appraiser

PR #339 introduces moving averages for metrics, categories and assets in the Appraiser. There are getters defined for each, which can be integrated with. Note that moving averages become stale once the current timestamp is more than one frequency period ahead of the last observation. When fetching a moving average, it is important to notify the consumer if the data is stale, so that it would not use it unsafely. It could result in arbitrage opportunities in upstream code.

Centralization risks

TRST-CR-1 Certain permissioned roles have key privileges and must be trusted

The Olympus architecture has a layered permission structure. In the context of the RBSv2 upgrade, the PriceConfig, SupplyConfig and TreasuryConfig contracts are all managed by admins. If the admin accounts were to be compromised or were to go rogue, they could destabilize the system through many different vectors, to the extent of stealing most of the value of the treasury.

The BunniAdmin can also cause significant damage through depositing of significant amounts of the treasury into Bunni and setting the PRICE TWAP maliciously, so that it can be manipulated (for example with a large deviation BPS).

Systemic risks

TRST-SR-1 Integration risks

The current iteration of the codebase composes with several external contracts. These include:

- BunniHub (through a self-deployed version with the original codebase)
- Uniswap-v3 by extension of integrating with Bunni
- Silo, through the SiloSupply module
- Any external pricers like Chainlink, UniswapV2/V3 TWAP facilities

If any of the integrations were to be affected by an attack or a bug, the Olympus assets or functionality could be impacted as well.

TRST-SR-2 Configuration assumptions

The Olympus RBS effectiveness is largely determined by its operation. The modules must be carefully configured with sound economic modelling in mind. Inaccuracies or misconfigurations could severely affect the overall health of the system, for example:

- Overly permissive price feed configuration
- Insertion of easily manipulable assets into the treasury or SUPPLY
- Incorrect categorization of assets