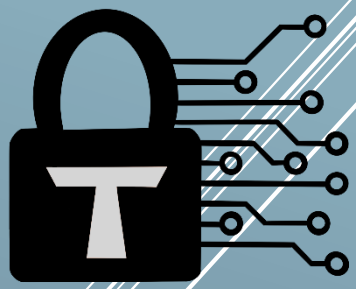
Trust Security



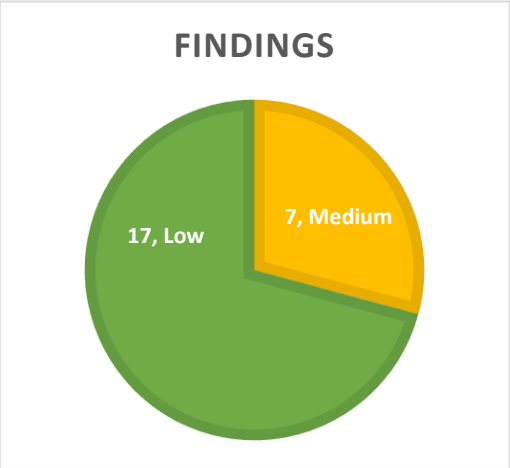
Smart Contract Audit

OlympusDAO Convertible Deposits

6th Nov 2025

Executive summary

Project Details



Category	Reserve
Audited file count	22
Lines of Code	3370
Auditor	carrotsmuggler, rvierdiiev, HollaDieWaldfee
Time period	15 Sept 2025 – 1 Nov 2025

Findings

Severity	Total	Open	Fixed	Acknowledged
High	0	0	0	0
Medium	7	0	5	2
Low	17	0	9	8

Centralization score

Signature

Digitally signed by Trust
Date: 2025.11.06 16:07:42+02'00'

Document properties

Versioning

Version	Date	Description
0.1	1st Nov 2025	Client report
0.2	6th Nov 2025	Mitigation review
0.3	6th Nov 2025	Second mitigation review

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

Files:

- src/bases/BaseAssetManager.sol
- src/bases/BasePeriodicTaskManager.sol
- src/external/clones/CloneERC20.sol
- src/libraries/AddressStorageArray.sol
- src/libraries/CloneableReceiptToken.sol
- src/libraries/DecimalString.sol
- src/libraries/ERC6909Wrappable.sol
- src/libraries/Timestamp.sol
- src/libraries/Uint2Str.sol
- src/modules/DEPOS/DEPOS.v1.sol
- src/modules/DEPOS/OlympusDepositPositionManager.sol
- src/modules/DEPOS/PositionTokenRenderer.sol
- src/policies/deposits/BaseDepositFacility.sol
- src/policies/deposits/ConvertibleDepositAuctioneer.sol
- src/policies/deposits/ConvertibleDepositFacility.sol
- src/policies/deposits/DepositManager.sol
- src/policies/deposits/DepositRedemptionVault.sol
- src/policies/deposits/ReceiptTokenManager.sol
- src/policies/EmissionManager.sol
- src/policies/Heart.sol
- src/policies/ReserveWrapper.sol
- src/proposals/ConvertibleDepositActivator.sol

Note for mitigation review:

- In addition to the changes that are directly related to the issues presented in the initial audit report, the mitigation review includes PR166, PR167 and PR168. Economic modelling of the auction mechanism remains out of scope. The risk arising from the design of the auction mechanism is highlighted in TRST-SR-3.

Repository details

- **Repository URL:** https://github.com/OlympusDAO/olympus-v3
- **Trusted commit hash:** 1ba7efd7db25dc90aad8792a5dad33a5697b137f
- **Commit hash:** 5b24625b0682cefdda2fa12da02337e17cc63929
- **Mitigation hash:** f0006fd2c4c3244ff157ec0d17d5d380b6949ff6
- **Mitigation hash #2:** be102178c0a092afe0ebccdf15d5a6474f35d37e

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

HollaDieWaldfee is a distinguished security expert with a track record of multiple first places in competitive audits. He is a Lead Auditor at Trust Security and Senior Watson at Sherlock.

carrotsmuggler is an SVM and EVM specialist, and competes in public audit contests on various platforms with 10+ Top 3 finishes.

rvierdiiev is a Web3 security researcher who participated in many public audit contests on multiple platforms and has a proven track record of experience

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**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly well documented.
Best practices	**Good**	Project mostly adheres to industry standards.
Centralization risks	**Good**	Project does not introduce significant unnecessary centralization risks.

Findings

Medium severity findings

TRST-M-1 Deposit restrictions can block loan repayments

- **Category:** Logical issues
- **Source:** BaseAssetManager.sol
- **Status:** Fixed

Description

The *BaseAssetManager* contract implements some restrictions in the *_depositAsset()* function.

```
if (amount_ < assetConfiguration.minimumDeposit) {
          revert AssetManager_MinimumDepositNotMet(

if (assetAmountBefore + amount_ > assetConfiguration.depositCap) {
              revert AssetManager_DepositCapExceeded(
```

Loan repayments through the *DepositManager* contract also invoke the same *_depositAsset()* function, placing the same restrictions. The issue is that this can block repayments, preventing loans from getting paid back if the loan amount is lower than the minimum deposit or if the deposit caps have been hit.

Recommended mitigation

Loan repayments should bypass deposit restrictions.

Team response

Fixed in PR152 and PR169.

Mitigation review

The issue has been fixed by bypassing both the minimum deposit and deposit cap check for loan repayments. It has been acknowledged that as a result the effective deposit cap is increased by the sum of all outstanding loans that have to be repaid.

TRST-M-2 Configuration changes can lead to unexpected losses

- **Category:** Race conditions
- **Source:** BaseDepositFacility.sol
- **Status:** Acknowledged

Description

When users call *reclaim()* on the deposit facility, their reclaimed amount depends on the admin configured reclaim rate, which can be as low as zero. The issue is that the user is unable

to specify an expected amount, so if the config changes while a user is trying to reclaim funds, they can end up with less funds than expected.

A similar issue exists in the *borrowAgainstRedemption()* function, where ***_assetFacilityMaxBorrowPercentages*** is used, which is also controlled by the admin, and the user is unable to specify an expected borrow amount.

Recommended mitigation

Allow the users to specify an expected amount in the *reclaim()* and *borrowAgainstRedemption()* functions.

Team response

Acknowledged. There are preview functions that will inform the caller of how much they will get back. The preview call would be made in the frontend just beforehand.

TRST-M-3 Tick decay is linear instead of exponential

- **Category:** Math issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Fixed

Description

The **tickSize** is expected to decrease based on the amount of OHM that has been converted on that day. The issue is that while this is expected to decay exponentially, the **tickSize** is only decreased by the multiplier linearly.

```
// Otherwise the tick size is halved as many times as the multiplier
newTickSize = auctionParams_.tickSize / (multiplier * 2);
```

Recommended mitigation

Change the formula to raise **2** to the power of **multiplier** instead of multiplying by it.

Team response

Fixed in PR151 and modified in PR168.

Mitigation review

As recommended, an exponential decay of the tick size has been implemented. Rather than a constant basis of **2**, it is now configurable in the range of **1-10**. There is an early return for when **multiplier > MAX_RPOW_EXP && _tickSizeBase != 1e18**, which is to avoid a revert inside the *rpow()* function. However, this has the side effect that even for a relatively small **_tickSizeBase** like **1.01e18**, for which a **multiplier > 41** still has a reasonable **divisor** value, *_getNewTickSize()* returns **_TICK_SIZE_MINIMUM=1**. Given that such a high multiplier means the day target has been reached more than 41 times, it could be considered a design decision to effectively stop the auction at that point by setting the tick size to the minimum. Overall, the issue of not implementing the exponential decay is mitigated. Furthermore, the early return for **divisor == 0** is unreachable since **divisor >= WAD**.

TRST-M-4 OHM target of the day can be exceeded without updating the tickSize

- **Category:** Logical issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Fixed

Description

During the auction, the **tickSize** decreases if the OHM target has been met. In the contract, this update happens via the *_getNewTickSize()* function, which is executed only if the tick capacity has been depleted.

```
if (output.tickCapacity <= convertibleAmount) {
    convertibleAmount = output.tickCapacity;
    // The tick has also been depleted, so update the price
    output.tickPrice = _getNewTickPrice(output.tickPrice, _tickStep);
    output.tickSize = _getNewTickSize(
        dayState.convertible + convertibleAmount + output.ohmOut,
        auctionParams);
```

The issue is that the global OHM target could have been crossed even if the ticks have not been depleted yet, since multiple deposit periods are allowed. This allows bidders to exceed the day capacity of OHM without updating the **tickSize**.

Recommended mitigation

Update the **tickSize** as soon as the OHM limit has been crossed instead of relying on capacity depletion.

Team response

Fixed in PR150.

Mitigation review

The mitigation that has been implemented goes beyond the suggestion of adjusting **tickSize** whenever a multiple of **target** is reached. Reaching a new multiple now also leads to a **price** increase in all deposit periods. This coupling between **target** and **price** is new and presents a deviation from the old auction mechanism. Since no specification for the new mechanism has been provided, its correctness against a specification can't be evaluated. Still, two consequences of the new mechanism can be observed which lead to the conclusion that the issue is not properly mitigated.

Firstly, since the depletion of a tick's **capacity** and the reaching of a **target** multiple can be arbitrarily close in terms of convertible OHM amounts, price changes become less predictable. For example, a tick's **capacity** might be depleted for a convertible amount of 10 OHM and the **target** multiple may be reached for a convertible amount of 10.0001 OHM. It is unclear why there should be two **price** increases with such a small gap in the convertible amount.

Secondly, *_synchronizePriceIncrease()* sets **tick.capacity = _currentTickSize** for the remaining deposit periods that are synchronized. This leads to another undefined dynamic where an interaction with one deposit period increases the capacity of the remaining deposit periods. Previously, only the opposite was possible. One deposit period could decrease

_currentTickSize and thus decrease the **capacity** of other deposit periods. Given that the fix has not mitigated the previous issue in the context of the old mechanism, but implemented a new mechanism with the above questionable consequences, the status of the finding is left "Open".

It is recommended to arrive at a specification of the auction mechanism against which the implementation can be checked. Evaluating the implementation of the specified mechanism and the mechanism itself are separate. The issue may also be mitigated with the understanding that the previous mechanism was correct, i.e., by adjusting **tickSize** (but not price) as recommended.

Team response

Fixed in PR173 and PR177.

Mitigation review

PR173 reverts part of the changes implemented in PR150. The behavior that has been implemented is such that reaching multiples of the day **target** behaves the same as depleting a tick's **capacity**, i.e., the **tickPrice** increases and **tickCapacity** is refilled up to **tickSize**. The difference between reaching the day **target** and depleting a tick's **capacity** is that the former updates **tickSize** while the latter does not. This behavior has been specified by the team as intended and the implementation has been checked to comply with the specification.

TRST-M-5 Update for _currentTickSize is not applied correctly

- **Category:** Logical issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Fixed

Description

In the *_bid()* function, if the **_currentTickSize** has changed, the *_getCurrentTick()* function is invoked when updating the other deposit periods. In this *_getCurrentTick()* function, there is an early return in case the last update has occurred in the same block.

```
uint256 timePassed = block.timestamp - previousTick.lastUpdate;
uint256 capacityToAdd = (_auctionParameters.target * timePassed) /
1 days /
_depositPeriods.length();
// Skip if the new capacity is 0
if (capacityToAdd == 0) return previousTick;
```

This allows tick size updates to be skipped when there are updates in multiple deposit periods within the same block.

If there are two bids for periods 1 and 2 in the same block, and later within the same block for period 2, the output **tickSize** is decreased so that **_currentTickSize** is now outdated. But the internal *_getCurrentTick()* call will return early since it's in the same block. So, the **tickSize** for period 1 will not get updated, even though the **tickSize** for period 2 has changed, leading to an incomplete update.

Recommended mitigation

Allow the tick capacity to be updated even when calling multiple times in the same block.

Team response

Fixed in PR150.

Mitigation review

Verified, the recommendation has been implemented.

TRST-M-6 Tick capacity and price changes can be griefed

- **Category:** Griefing issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Acknowledged

Description

In the *_getCurrentTick()* function, there are two checks present. First, it checks if the **newCapacity** exceeds the default **tickSize**, and then it checks if the updated **newCapacity** exceeds **_currentTickSize**.

```
uint256 tickSize = _auctionParameters.tickSize;
while (newCapacity > tickSize) {
//…
tick.capacity = newCapacity > _currentTickSize ? _currentTickSize : newCapacity;
```

This leads to a situation where if the **newCapacity** calculated is larger than **_currentTickSize** but lower than **_auctionParameters.tickSize**, neither the price nor the **tickSize** is adjusted.

Assume **_currentTickSize = 100**, **_auctionParameters.tickSize = 200** and currently the **tick.capacity** is only **80**. The capacity will gradually grow over time from **80** to **100** and then stop. This is because if the **newCapacity** is calculated as **120**, it doesn't exceed the default **tickSize** of **200**, so the price isn't changed, and it is also capped by the **_currentTickSize**. So, the auction remains at the same price and **tickSize** until the **newCapacity** hits **200**.

Furthermore, users can still bid and buy in this state, and intentionally grief the auction to keep it in this range indefinitely.

Recommended mitigation

The pricing algorithm needs to be re-evaluated in order to fix this issue. It should not be possible for the capacity to be clamped in such a way that even as time passes the tick is effectively stale.

Team response

Acknowledged. This would only occur if the day target has been exceeded (**currentTickSize < auctionParameters.tickSize**), so the impact is likely to be less.

Mitigation review

It is agreed that the likelihood depends on whether the day target has been reached as otherwise **_currentTickSize == auctionParamters.tickSize** which prevents the issue. But

exceeding the day target is core functionality, and there is no clear mechanism or reasoning why it should be assumed that the day target won't be exceeded or is unlikely to be exceeded.

TRST-M-7 Repayments can leak assets which causes DOS

- **Category:** DOS issues
- **Source:** DepositManager.sol
- **Status:** Fixed

Description

In the *borrowingRepay()* function, the repaid funds are deposited in the underlying vault. However, the output of this deposit action is not used, and instead the passed in **params_.amount** value is used for accounting.

```
// We ignore the actual amount deposited into the vault, as the payer will not be able
to over-pay in case of an off-by-one issue
// This takes place before any state changes to avoid ERC777 re-entrancy
_depositAsset(params_.asset, params_.payer, params_.amount);
//...
return params_.amount;
```

The issue is that **params_.amount** is not the actual amount credited to the vault. It can be underpaid due to rounding down during share conversion. So, there is a possible revert when the solvency is checked in the *_validateOperatorSolvency()* function.

```
(, uint256 depositedSharesInAssets) = getOperatorAssets(asset_, operator_);
bytes32 assetLiabilitiesKey = _getAssetLiabilitiesKey(asset_, operator_);
uint256 operatorLiabilities = _assetLiabilities[assetLiabilitiesKey];
uint256 borrowedAmount = _borrowedAmounts[assetLiabilitiesKey];
if (operatorLiabilities > depositedSharesInAssets + borrowedAmount) {
    revert DepositManager_Insolvent(
```

This is because even though the vault is credited **params_.amount** number of assets, the solvency only counts **previewRedeem(shares)** number of assets, which can be off by a small amount due to rounding.

Thus, this solvency check can revert and stop repayments for small loan repayments because even though the user borrowed **X** and repaid **X** tokens, the vault only sees their assets go back up by **X - roundingError** due to share conversion.

An attacker can also intentionally set up the protocol to hit such reverts by first claiming yield, performing multiple repays, and leaving it in a state where the next repayment by any user will trigger the revert due to the rounding difference.

Recommended mitigation

Debt repayments must be implemented so that the repaid amount is overpaid rather than underpaid. This means a refactoring is necessary since for now the protocol only supports a deposit flow where assets are underpaid. In other words, when a repayment deposits **amount** of assets into the vault which corresponds to **shares** number of shares, only **previewRedeem(shares)** must be credited to the repayment of the loan **principal**.

Team response

Fixed in PR152.

Mitigation review

The mitigation introduces an issue since *DepositManager* and *BaseDepositFacility* rely for their accounting on **borrowedAmounts_** and **assetCommittedDeposits_** respectively. A loan with a certain **principal** must only subtract from **borrowedAmounts_** and add to **assetCommittedDeposits_** the same amount of **principal** and not more.

A loan with **principal** increases **_borrowedAmounts** in *DepositManager.borrowingWithdraw()* by **principal**. But due to the mitigation, **borrowedAmounts_** can be reduced by more than **principal** in *DepositManager.borrowingRepay()*. The impacts are multiple, and it is sufficient to point out *DepositManger.borrowingDefault()* where a revert due to underflow in **_borrowedAmounts** can occur, making it impossible to default loans.

To mitigate the issue, it must be recognized that the intermediary *DepositManager* and *BaseDepositFacility* require for their accounting that **principalRepaid <= principalWithdrawn**. The finding can thus be mitigated by passing through a **maxPrincipal** variable from *DepositRedemptionVault.repayLoan()* which is set to equal the remaining principal. This variable must then be used downstream to limit what is subtracted from **borrowedAmounts_** and what is added to **assetCommittedDeposits_**. Furthermore, it is recommended to implement test cases to ensure the solvency variants are maintained in the new repayment flow. The suggested mitigation is outlined in the code snippets below.

DepositRedemptionVault.repayLoan() must pass the maximum **principal** to repay.

```
diff --git a/src/policies/deposits/DepositRedemptionVault.sol
b/src/policies/deposits/DepositRedemptionVault.sol
index 4b75b365..2a03ffcd 100644
--- a/src/policies/deposits/DepositRedemptionVault.sol
+++ b/src/policies/deposits/DepositRedemptionVault.sol
@@ -727,6 +727,7 @@ contract DepositRedemptionVault is Policy,
IDepositRedemptionVault, PolicyEnable
                IERC20(redemption.depositToken),
                redemption.depositPeriod,
                principalToRepay,
+               loan.principal,
                address(this)
            );
```

This is used in *DepositManager.borrowingRepay()* and *BaseDepositFacility.handleLoanRepay()*.

```
diff --git a/src/policies/deposits/BaseDepositFacility.sol
b/src/policies/deposits/BaseDepositFacility.sol
index 4c2bf1d1..55004181 100644
--- a/src/policies/deposits/BaseDepositFacility.sol
+++ b/src/policies/deposits/BaseDepositFacility.sol
@@ -305,6 +305,7 @@ abstract contract BaseDepositFacility is Policy, PolicyEnabler,
IDepositFacility
        IERC20 depositToken_,
        uint8,
        uint256 amount_,
+       uint256 maxPrincipal,
        address payer_
```

```
    ) external nonReentrant onlyEnabled onlyAuthorizedOperator returns (uint256) {
        // Process the repayment through DepositManager
@@ -314,6 +315,7 @@ abstract contract BaseDepositFacility is Policy, PolicyEnabler,
IDepositFacility
            IDepositManager.BorrowingRepayParams({
                asset: depositToken_,
                payer: payer_,
+               maxPrincipal: maxPrincipal,
                amount: amount_
            })
        );
@@ -321,8 +323,8 @@ abstract contract BaseDepositFacility is Policy, PolicyEnabler,
IDepositFacility
        // Repayment of a principal amount increases the committed deposits (since it
was deducted in `handleBorrow()`
        _assetOperatorCommittedDeposits[
            _getCommittedDepositsKey(depositToken_, msg.sender)
-       ] += repaymentActual;
-       _assetCommittedDeposits[depositToken_] += repaymentActual;
+       ] += repaymentActual > maxPrincipal ? maxPrincipal : repaymentActual;
+       _assetCommittedDeposits[depositToken_] += repaymentActual > maxPrincipal ?
maxPrincipal : repaymentActual;

        return repaymentActual;
    }
diff --git a/src/policies/deposits/DepositManager.sol
b/src/policies/deposits/DepositManager.sol
index c1c1e3e2..78c659a3 100644
--- a/src/policies/deposits/DepositManager.sol
+++ b/src/policies/deposits/DepositManager.sol
@@ -738,8 +738,8 @@ contract DepositManager is Policy, PolicyEnabler, IDepositManager,
BaseAssetMana
        // Update borrowed amount
        // Reduce by the actual amount, to avoid leakage
        // But cap at the borrowed amount, to avoid an underflow if there is an over-
payment
-       _borrowedAmounts[borrowingKey] -= currentBorrowed < actualAmount
-           ? currentBorrowed
+       _borrowedAmounts[borrowingKey] -= params_.maxPrincipal < actualAmount
+           ? params_.maxPrincipal
            : actualAmount;

        // Validate operator solvency after borrowed amount change
```

Team response

Fixed in PR174.

Mitigation review

Verified, the recommendation has been implemented, and test cases have been added.

Low severity findings

TRST-L-1 setTickStep() sets tickStep retroactively

- **Category:** Logical issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Fixed

Description

The function *setTickStep()* updates the tick step used when changing the price. The issue is that the **lastUpdate** value is not updated to the current timestamp. Since this timestamp is not recorded, changes in the price will incorporate this new **tickStep** retroactively.

Recommended mitigation

Record the **lastUpdate** value when changing **tickStep** by calling *updateCurrentTicks(0)*.

Team response

Fixed in PR157.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-2 EmissionManager can get out of sync when disabled

- **Category:** Logical issues
- **Source:** EmissionManager.sol
- **Status:** Acknowledged

Description

When *_disable()* is called in the *EmissionManager* contract, it calls *setAuctionParameters()* with **0** values to disable auctions.

```
cdAuctioneer.setAuctionParameters(0, 0, 0);
```

The issue is that this takes up an **_auctionResultsNextIndex** slot. When enabled again via *restart()* or *enable()*, the next *execute()* function call will again call *setAuctionParameters()* and then check if the tracking period is over or not. However, due to the auction parameter setting in the *disable()* function, the **0** index could be completely skipped, preventing the evaluation of underselling of OHM, since **cdAuctioneer.getAuctionResultsNextIndex() == 0** will return **false** even though the tracking period has already ended.

Recommended mitigation

When the emission manager is disabled and restarted/enabled, it must be ensured that the emissions are picked up in a consistent manner.

Team response

Acknowledged.

TRST-L-3 EmissionManager uses stale values while auctioneer is paused

- **Category:** Logical issues
- **Source:** EmissionManager.sol
- **Status:** Fixed

Description

If the auctioneer is paused, subsequent calls to *setAuctionParameters()* do not increment **_auctionResultsNextIndex** and instead return early due to a check.

```
if (!isEnabled) return;
```

Thus **_auctionResultsNextIndex** is not incremented and the tracking stops, so the *EmissionManager* keeps using stale values to check for underselling. If the auctioneer is stopped in a state where **_auctionResultsNextIndex = 0**, bond markets may get created each day.

Recommended mitigation

Both the auctioneer and the *EmissionManager* contracts can be disabled independently, which increases the risk that the components can get out of sync. Since *ConvertibleDepositAuctioneer* is a dependency of *EmissionManager*, a possible solution is to make *ConvertibleDepositAuctioneer* configurations go through *EmissionManager* entirely, so that a synced state can be maintained.

Team response

Fixed in PR170.

Mitigation review

The issue has been addressed in *EmissionManager.execute()* by expanding the **cdAuctioneer.getAuctionResultsNextIndex() == 0** condition with a check that **IEnabler(address(cdAuctioneer)).isEnabled().** This ensures that there is no bond market creation based on outdated information from the auctioneer and mitigates the risk of over-selling OHM. Still, by skipping bond market creation altogether, the consideration of under-selling remains. And it also remains an error-prone design that *EmissionManager* and *ConvertibleDepositAuctioneer* can be enabled / disabled and managed separately. This issue has been marked as “fixed” since the economic risk is about over-selling OHM, and this has been mitigated.

TRST-L-4 Bond market creation can be griefed by executing heartbeat with specific gas limit

- **Category:** Gas issues
- **Source:** EmissionManager.sol
- **Status:** Acknowledged

Description

When OHM underselling is detected, the *execute()* function in *EmissionManager* tries to create a bond market.

```
if (difference < 0) {
    uint256 remainder = uint256(-difference);
    // Attempt to create the bond market
    try this.createPendingBondMarket() {
        // Do nothing if successful
        // createPendingBondMarket() resets the pending capacity, so it does not
need to be done here
    } catch {
        // We don't want the periodic task to fail, so catch the error
        // But trigger an event that can be monitored
        // Upon failure, the createPendingBondMarket() function can be called again
to trigger the bond market
        emit BondMarketCreationFailed(remainder);
    }
}
```

Since a try-catch is used, only 63/64 of the available gas is forwarded for the function call. If this amount of gas is insufficient, the market creation will fail. However, if the remaining 1/64 amount of gas is sufficient for the transaction to succeed, the *Heart.beat()* execution can still finish successfully without reverting. This can create a minor DOS since the bond market needs to be created manually by the admin or manager.

Due to the gas restrictions of this attack vector, this can only happen in a state where there are **0** pending rewards to minimize the gas required for the transaction to succeed and to skip the minting of OHM rewards to the keeper.

Recommended mitigation

Ensure the try/catch block is called with sufficient gas which ensures successful bond market creation.

Team response

Acknowledged.

TRST-L-5 Setting newCapacity at minPrice is inconsistent

- **Category:** Logical issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Fixed

Description

If the tick price is too low, it is capped to a minimum value.

```
if (tick.price < _auctionParameters.minPrice) {
        tick.price = _auctionParameters.minPrice;
        newCapacity = tickSize;
        break;
    }
}
```

```
// Set the capacity, but ensure it doesn't exceed the current tick size
// (which may have been reduced if the day target was met)
tick.capacity = newCapacity > _currentTickSize ? _currentTickSize : newCapacity;
```

If **newCapacity** was calculated to be higher than **tickSize** (prior to the **newCapacity = tickSize** assignment), the last line will already cap it to the value of **_currentTickSize**, which itself is **<= tickSize**. Therefore, in this case **newCapacity = tickSize** is redundant. The second possibility is that **newCapacity < tickSize**, and so **newCapacity** is increased to **tickSize**. This gives the **minPrice** boundary a meaning that there is infinite capacity. However, even at the **minPrice** the same gradual capacity increase applies like at other prices. Thus, it is inconsistent to bring up **newCapacity** to **tickSize**.

If **minPrice** is profitable to make bids at, the price naturally moves up, so in practice this **tick.price < _auctionParameters.minPrice** condition is not expected to materially influence auction outcomes.

Recommended mitigation

Remove the direct assignment of **newCapacity** to the **tickSize**.

Team response

Fixed in PR160.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-6 Small redemptions and default claims can revert

- **Category:** Rounding issues
- **Source:** DepositRedemptionVault.sol
- **Status:** Fixed

Description

In the *claimDefaultedLoan()* and *finishRedemption()* functions, even small withdrawals of a few wei are attempted to be withdrawn.

```
if (retainedCollateral > 0) {
    retainedCollateralActual = IDepositFacility(redemption.facility)
        .handleCommitWithdraw(
            IERC20(redemption.depositToken),
            redemption.depositPeriod,
            retainedCollateral,
            address(this)
        );
}
```

The issue is that small withdrawals of a few wei revert in the *BaseAssetManager* contract due to a **1 wei** shares minimum.

```
shares = vault.convertToShares(amount_);
// Amount of shares must be non-zero
```

```
if (shares == 0) revert AssetManager_ZeroAmount();
```

Thus, small redemption finalizations, or small loan default claims, can revert, which can be unexpected for integrators.

Recommended mitigation

Consider either skipping the withdrawals for dust amounts, since rounding down the funds that the user gets to zero is just like rounding any other amount, or document this behavior. It must be noted that even if the OlympusDAO protocol is not the source of the revert, the external vault may revert instead if it implements zero amount checks on its own.

Team response

Fixed in PR152.

Mitigation review

The mitigation is insufficient to ensure that the call to **vault.redeem()** does not revert. Suppose the vault's exchange rate is such that 1 asset is worth 1.5 shares. *BaseAssetManager* performs the following calculations:

```
            shares = vault.convertToShares(amount_);


            // Early exit if the amount of shares is 0, to prevent a revert
            if (shares == 0) return (0, 0);


            assetAmount = vault.redeem(shares, depositor_, address(this));
```

If **amount_ = 1**, then **shares = 1** since *convertToShares()* rounds down. Then, since *vault.redeem()* rounds down a second time, **shares = 1** corresponds to **assetAmount = 0** which can revert inside the vault.

It is recommended to check with a call to *previewRedeem()* whether the redeemed **assetAmount** is greater zero:

```
shares = vault.convertToShares(amount);

if (shares == 0) return (0, 0);

previewAmount = vault.previewRedeem(shares);

if (previewAmount == 0) return (0, 0);
```

Team response

Fixed in PR175.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-7 No incentive to liquidate small loans

- **Category:** Logical issues
- **Source:** DepositRedemptionVault.sol
- **Status:** Acknowledged

Description

Keepers call *claimDefaultedLoan()* on defaulted loans to liquidate them and earn a reward while keeping the overall protocol healthy. The issue is that since there is no minimum cap on the loan limit and the keeper incentives are a fixed percentage of the retained collateral, small loans can have very low incentives, which might not even cover the gas cost of liquidating the position.

Thus, for small loan sizes, there are not enough incentives to attract users/bots to act as keepers and keep the platform healthy overall. Such small loans need to be liquidated by the admin/team themselves.

Recommended mitigation

Consider either putting a minimum limit on loans or, equally, on retained collateral, to ensure sufficient keeper incentives, or document this behavior.

Team response

Acknowledged.

TRST-L-8 previewConvert() might not match convert() calls due to duplicate position IDs

- **Category:** Logical issues
- **Source:** ConvertibleDepositFacility.sol
- **Status:** Acknowledged

Description

The *previewConvert()* function is meant to exactly replicate the results of *convert()*. The issue is that while the *convert()* function updates the position by calling *DEPOS.setRemainingDeposit()*, *previewConvert()* does not account for it. Thus, if the passed in **positionIds_** array contains the same ID multiple times, the *previewConvert()* function will give incorrect results since it will assume the full **remainingDeposit** for multiple conversions instead of updating the **remainingDeposit** in between conversions.

Recommended mitigation

Consider disallowing duplicates in the input to *previewConvert()* and *convert()*.

Team response

Acknowledged.

TRST-L-9 ConvertibleDepositFacility reverts instead of failing softly

- **Category:** Logical issues
- **Source:** ConvertibleDepositFacility.sol
- **Status:** Fixed

Description

The *execute()* function in the *ConvertibleDepositFacility* contract is not meant to fail loudly, since yield claiming is not critical to the system, as indicated by the comment. However, the function reverts, reverting the entire transaction, instead of just emitting an event.

```
// This avoids the periodic task from failing loudly, as the claimAllYield function
is not critical to the system
revert CDF_ClaimAllYieldFailed();
```

Recommended mitigation

Replace the revert with an event emission.

Team response

Fixed in PR153.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-10 ReserveWrapper doesn't check if treasury is active

- **Category:** Logical issues
- **Source:** ReserveWrapper.sol
- **Status:** Fixed

Description

The *execute()* function in the *ReserveWrapper* contract implements a few checks to ensure it doesn't revert and block other period tasks as shown below.

```
if (reserveBalance == 0) {
    return;
}
// Skip if depositing the balance would result in zero shares (as we don't want
this to revert)
if (_SRESERVE.previewDeposit(reserveBalance) == 0) {
    return;
}
```

However, it does not check if the treasury is active before calling *withdrawReserves()* on it. This treasury function has the **onlyWhileActive** modifier, and this call will thus fail, blocking other period tasks if the treasury is inactive.

Recommended mitigation

Add a check to ensure the treasury is active before calling it.

Team response

Fixed in PR154.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-11 getWrappableTokens() is vulnerable to gas griefing due to permissionless token creation

- **Category:** Griefing issues
- **Source:** ReceiptTokenManager.sol
- **Status:** Acknowledged

Description

Anyone can call *createToken()* in the *ReceiptTokenManager* contract and create a wrappable token. The issue is that if many such tokens are created, calling *getWrappableTokens()* on-chain will become impossible, since it is forced to iterate over a practically infinite array.

Recommended mitigation

Consider adding some access control to the *createToken()* function.

Team response

Acknowledged.

TRST-L-12 Position NFTs are prone to re-org issues due to incremental ID

- **Category:** Re-org issues
- **Source:** OlympusDepositPositionManager.sol
- **Status:** Acknowledged

Description

Position NFTs are minted to users with IDs based on the **_positionCount** counter. This is unlike the receipt token, whose **positionId** is calculated from the hash of certain parameters. The issue with a linearly increasing **positionId** is that in case of a re-org, the user could have received some **tokenId**, but they could have referenced some other **tokenId** in subsequent transactions.

Say the user receives NFT IDs 5,6,7,8 in bids, and splits the **tokenId** 7. Due to a chain re-org, their bids could be placed in a different position, and they could have instead minted NFT IDs 6, 7, 8, 9, but they would still be splitting **tokenId** 7. So in the first case, they split their 3rd token, while in the second case, they split their 2nd token. Thus, the user cannot reliably control their NFT IDs, since re-orgs determine what token IDs they get.

Post-merge, the above considerations become relevant if OlympusDAO is deployed on Layer2 chains on which deep re-orgs are possible.

Recommended mitigation

Consider implementing a system like in the receipt token, where the **positionId** IDs are not incremented linearly but instead calculated by hashing a subset of parameters. To disambiguate NFTs with the same parameters, a per-owner incremental salt may be used. Alternatively, the finding may be acknowledged.

Team response

Acknowledged.

TRST-L-13 EIP165 compliance requires to return true for EIP165 interface

- **Category:** EIP compliance issues
- **Source:** BaseAssetManager.sol
- **Status:** Fixed

Description

Certain contracts implement the EIP165 *supportsInterface()* function, but don't follow the EIP standard. According to EIP165, **supportsInterface(0x01ffc9a7)** must return **true**. Since the current contracts only check for the **interfaceId** of the contract itself and not the EIP165 **interfaceId** as well, interfaces will not be discoverable by contracts checking for EIP165 support.

Recommended mitigation

Return **true** for **0x01ffc9a7**.

Team response

Fixed in PR163.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-14 Single failed yield claim blocks all yield claims

- **Category:** Logical issues
- **Source:** ConvertibleDepositFacility.sol
- **Status:** Acknowledged

Description

The *execute()* function in the deposit facility uses a try-catch statement to claim the yield. The issue is that it uses the try-catch in the outermost call.

```
 try this.claimAllYield() {
            // Do nothing
        } catch {
            // This avoids the periodic task from failing loudly, as the
claimAllYield function is not critical to the system
            revert CDF_ClaimAllYieldFailed();
        }
```

This means that if a single yield claim fails, the entire yield claim fails. Yield must then be claimed manually for each configured asset separately.

Recommended mitigation

Consider using try-catch for each individual yield claim instead of all the claims at once.

Team response

Acknowledged. There is a fallback function for claims on individual assets.

TRST-L-15 createPendingBondMarket() doesn't check if EmissionManager is enabled

- **Category:** Privilege escalation issues
- **Source:** EmissionManager.sol
- **Status:** Fixed

Description

The *createPendingBondMarket()* function in the *EmissionManager* contract does not check if the contract itself is enabled or not. This is a privilege escalation issue, since the manager role is not allowed to enable the contract, but due to the lack of this check, they can create bond markets while the contract is disabled.

Recommended mitigation

In the *createPendingBondMarket()* function, check **isEnabled**.

Team response

Fixed in PR155.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-16 Auctioneer can consume excessive gas due to iterative algorithms

- **Category:** Gas issues
- **Source:** ConvertibleDepositAuctioneer.sol
- **Status:** Acknowledged

Description

The *ConvertibleDepositAuctioneer* uses two loops to iterate over **remainingDeposit** and **newCapacity**.

```
while (newCapacity > tickSize) {
    //...
}
//...
while (remainingDeposit > 0) {
    //...
}
```

In both cases the gas consumption of the loops is not limited by any restrictions in the smart contract and the only relevant restriction defined is that of **_TICK_SIZE_MINIMUM = 1**. This would mean there is one loop iteration in *_previewBid()* for each wei of OHM. Furthermore, there exists no documentation to derive safe auction parameters from, so that in practice, only excessive gas consumption or DOS can be used as a signal to change the auction parameters.

If such a small value of tick size is used, the gas required for the loop can incur significant costs for bidders and might even cross the block gas limit, blocking auctions until a reconfiguration happens. Note that the call to *setAuctionParameters()* that the *EmissionManager* makes may not be able to resolve the problem due to the dependence on bringing up the current ticks.

```
function setAuctionParameters(
    uint256 target_,
    uint256 tickSize_,
    uint256 minPrice_
) external override onlyRole(ROLE_EMISSION_MANAGER) {
    uint256 previousTarget = _auctionParameters.target;
    // Update tick state for enabled assets and periods
    // This prevents retroactive application of new parameters
    _updateCurrentTicks(0);
```

So, the reconfiguration would require disabling and enabling the auctioneer, erasing auction results in the process.

Recommended mitigation

The issue can be mitigated in two ways. One is to define the assumptions under which the gas consumption of the auctioneer remains acceptable, and to document them or to implement bounds within the auctioneer. The other possible mitigation is to change the auctioneer accounting in such a way that *_previewBid()* and *_getCurrentTick()* only need constant time.

Team response

Acknowledged.

TRST-L-17 Pending bond market capacity is not reset when scaledCapacity==0

- **Category:** Logical issues
- **Source:** EmissionManager.sol
- **Status:** Fixed

Description

In PR167, a scalar for bond market capacity has been introduced, and a bond market is only created when **scaledCapacity>0**. Otherwise, no action is performed. If **scaledCapacity==0**, **bondMarketPendingCapacity** remains at its previous value which allows to create the pending bond market for more than one auction tracking period. This is against the specification.

Recommended mitigation

Add an **else** case for when **scaledCapacity==0** which sets **bondMarketPendingCapacity=0**.

Team response

Fixed in PR176.

Mitigation review

Verified, the recommendation has been implemented.

Additional recommendations

TRST-R-1 Remove redundant modifier

ReceiptTokenManager.mint() has the **onlyValidTokenId** modifier. The function later calls *_mint()*, which also has the same modifier. It is recommended to remove the double check from *ReceiptTokenManager.mint()*.

The same situation occurs with *ReceiptTokenManager.burn()* and *_burn()* functions.

TRST-R-2 Minimum deposit check is not applied to startRedemption() function

The *startRedemption()* function allows user to redeem any amount and leave the position with an amount less than the minimum deposit allowed. It is recommended to introduce the minimum deposit check after redemption is started.

TRST-R-3 PositionTokenRenderer renders unwrapped positions

Currently *PositionTokenRenderer* renders all positions of *IDepositPositionManager***.** As only a wrapped position has an NFT minted to it, it is reasonable to limit the renderer to work with wrapped positions only.

TRST-R-4 DecimalString has inconsistent trailing zeros

In the *DecimalString* library, when **bStr.length <= valueDecimals**, i.e., the number starts with **0.**, trailing zeros are included in the result. But otherwise, trailing zeros are not included. Including the trailing zeros is an error indicated by this comment.

So, **1230** with **valueDecimals=4** and **decimalPlaces=4** gives **0.1230** (not **0.123**!) while **1230** with **valueDecimals=3** and **decimalPlaces=4** gives **1.23** (correct, no trailing zeros).

Furthermore, *the* **i<bStr.length** check in the loop is not necessary.

```
for (uint256 i = 0; i < maxDecimalPlaces - zerosToAdd && i < bStr.length; i++) {
                smallResult[i + 2 + zerosToAdd] = bStr[i];
            }
```

This is because there are enough leading zeros, so there is no risk of iterating beyond the **bStr** length.

TRST-R-5 Error prone wrapped token approvals

In the *CloneableReceiptToken* contract, the *burnFrom()* function checks for allowance. Only the *ReceiptTokenManager* is allowed to call the *mintFor()* and *burnFrom()* functions, thus an allowance check is not necessary and is error-prone design.

In the current state, users don't need to give any allowance for wrapping but do need to give allowance to the *ReceiptTokenManager* in order to unwrap tokens. Furthermore, if functionality were to be added to allow users to unwrap each other's tokens, provided approvals are given, this code would have an issue since it only checks for approvals of the *ReceiptTokenManager* and not the actual initiator of the transaction.

A more robust design would be to pass the **msg.sender** from the *ReceiptTokenManager* context into this function, and applying an allowance check on that instead.

```
 diff --git a/src/libraries/CloneableReceiptToken.sol
b/src/libraries/CloneableReceiptToken.sol
index 21a632c0..49e408de 100644
--- a/src/libraries/CloneableReceiptToken.sol
+++ b/src/libraries/CloneableReceiptToken.sol
@@ -75,11 +75,13 @@ contract CloneableReceiptToken is CloneERC20,
IERC20BurnableMintable, IDepositRe
     ///
     /// @param from_ The address to burn tokens from
     /// @param amount_ The amount of tokens to burn
-    function burnFrom(address from_, uint256 amount_) external onlyOwner {
-        uint256 allowed = allowance[from_][msg.sender];
+    function burnFrom(address from_, uint256 amount_, address sender_) external
onlyOwner {
+        if (from_ != sender_) {
+            uint256 allowed = allowance[from_][sender_];

-        if (allowed != type(uint256).max) {
-            allowance[from_][msg.sender] = allowed - amount_;
+            if (allowed != type(uint256).max) {
+                allowance[from_][msg.sender] = allowed - amount_;
+            }
         }
```

TRST-R-6 previewConvert() should check convertible status first

The *previewConvert()* function in the *OlympusDepositPositionManager* contract can return **0** if a position has expired, even if the position is not actually convertible. This is because the expiry is checked first and returns **0** if expired, while the actual *_isConvertible()* check is performed later and reverts if not convertible.

Consider checking *_isConvertible()* first, so that the function does not erroneously return a **0**.

TRST-R-7 Redundant check when checking tokenData length

In the *ERC6909Wrappable* contract, there's a check on the token data length.

```
if (tokenData.length == 0) revert ERC6909Wrappable_InvalidTokenId(tokenId_);
```

This check is redundant and always passes since *getTokenData()* encodes at least two **bytes32** and a **uint8** type. Consider removing this.

TRST-R-8 Redundant auction results assignments

The *_enable()* function in the *ConvertibleDepositAuctioneer* contract makes assignments for the auction results.

```
_auctionResults = new int256[](_auctionTrackingPeriod);
_auctionResultsNextIndex = 0;
```

These are, however, redundant since the same assignment also happens inside *setAuctionTrackingPeriod()*. Consider removing the two lines from the *_enable()* function.

TRST-R-9 Price and distributor calls can be periodic tasks

In *Heart*, the calls to *PRICE.updateMovingAverage()* and *distributor.triggerRebase()* can be made periodic tasks.

In particular, this comment is invalid and was based on an earlier version of the code.

```
// This cannot be a periodic task, because it requires a policy with permission to
call the updateMovingAverage function
```

TRST-R-10 BaseDepositFacility should call _split() before DEPOS.split()

The *BaseDepositFacility* contract's *_split()* function is meant to be implemented by future implementations, for now it is empty. This internal *_split()* function is called inside the external *split()* function after *DEPOS.split()*.

```
 // Perform the split
// This will revert if the amount is 0 or greater than the position amount
uint256 newPositionId = DEPOS.split(positionId_, amount_, to_, wrap_);
// Allow inheriting contracts to perform custom actions when a position is split
_split(positionId_, newPositionId, amount_);
```

This pattern is unsafe since *DEPOS.split()* can give a callback from the NFT mint and therefore change the state in which *_split()* observes the position (e.g., transfer it somewhere else). Consider calling *_split()* before *DEPOS.split()*.

TRST-R-11 Optimization for uint2str() function

The *uint2str()* function can be slightly optimized with the following diff.

```
diff --git a/src/libraries/Uint2Str.sol b/src/libraries/Uint2Str.sol
index a28d399d..2791734e 100644
--- a/src/libraries/Uint2Str.sol
+++ b/src/libraries/Uint2Str.sol
@@ -18,9 +18,7 @@ function uint2str(uint256 _i) pure returns (string memory) {
     uint256 k = len;
     while (_i != 0) {
         k = k - 1;
-        uint8 temp = (48 + uint8(_i - (_i / 10) * 10));
-        bytes1 b1 = bytes1(temp);
-        bstr[k] = b1;
+        bstr[k] = bytes1(48 + uint8(_i % 10));
         _i /= 10;
     }
     return string(bstr);
```

TRST-R-12 Unreachable code in auctioneer _previewbid()

In the *_previewBid()* function, there is an if-else statement setting the remaining deposit.

```
if (depositAmount <= remainingDeposit) {
    remainingDeposit -= depositAmount;
} else {
    remainingDeposit = 0;
}
```

This else clause is unreachable. This is because the code first does a *mulDiv()* which rounds down and then a *mulDivUp()* with **remainingDeposit** and **tickPrice**. Due to this, **depositAmount** is always guaranteed to be at most **remainingDeposit**.

Consider adding a comment in the else clause to document that this is never expected to be hit.

TRST-R-13 Different deposit periods have same minPrice

The same **minPrice** is used for all deposit periods. This is not ideal, since options with different expiry should have different prices and longer running call options should be more expensive.

Given the pricing risk of the OHM conversion options that has been pointed out in TRST-SR-3, which is accepted, using an overall **minPrice** is a design choice. Nonetheless, it is expected that market values for different deposit periods diverge, and thus separate **minPrices** per deposit period would allow for a more nuanced control of auction profitability.

TRST-R-14 previewExtendLoan() doesn't check loan state

The *extendLoan()* function in the *DepositRedemptionVault* contract checks the state of the loan.

```
if (loan.dueDate == 0) revert RedemptionVault_InvalidLoan(msg.sender,
redemptionId_);
// Validate that the loan is not:
// - expired
// - defaulted
// - fully repaid
if (block.timestamp >= loan.dueDate || loan.isDefaulted || loan.principal == 0)
    revert RedemptionVault_LoanIncorrectState(msg.sender, redemptionId_);
```

These checks are missing from the *previewExtendLoan()* function, so the preview function can return values even if the actual loan extension call is expected to revert.

TRST-R-15 Timestamp casting might overflow

In the *Timestamp* contract, the **uint48** value is cast into **int48**. The behavior that results is that as **timestamp** increases beyond **type(int48).max**, the returned year, month and day become undefined. Consider casting to **uint256** instead.

```
diff --git a/src/libraries/Timestamp.sol b/src/libraries/Timestamp.sol
index 9575fa64..4d758c2d 100644
--- a/src/libraries/Timestamp.sol
+++ b/src/libraries/Timestamp.sol
@@ -14,7 +14,7 @@ library Timestamp {
         uint256 month;
         uint256 day;
         {
-            int256 __days = int256(int48(timestamp) / 1 days);
+            int256 __days = int256(uint256(timestamp) / 1 days);

             int256 num1 = __days + 68_569 + 2_440_588; // 2440588 = OFFSET19700101
             int256 num2 = (4 * num1) / 146_097;
```

Furthermore, the original *BokkyPooBahsDateTimeLibrary*, which this is forked from, only supports dates up to the year **2345**, while the code here also handles years exceeding **10000**.

```
string memory yearStr = uint2str(year % 10_000);
```

Consider clamping the input to the year **2345** as anything beyond can be considered irrelevant.

Systemic risks

TRST-SR-1 Less frequent updates of Heart.beat() affect EmissionManager

The *Heart.beat()* function is designed to be called by keepers every 8 hours. If keepers fail to perform this task, updates may occur less frequently, which impacts the *EmissionManager*, since it is expected to be updated once per day. In this case, more convertible amount may be assigned to a day than was actually sold in that day, preventing the creation of the bond market even if there was underselling.

TRST-SR-2 Losses in the ERC4626 vault may cause insolvency in DepositManager

DepositManager deposits all funds into an ERC4626 vault if it is configured. It's possible that the vault faces a loss, which then decreases *getOperatorAssets()* amount. In this case *_validateOperatorSolvency()* may revert if amount is smaller than **operatorLiabilities.**

```
function _validateOperatorSolvency(IERC20 asset_, address operator_) internal view
{
        (, uint256 depositedSharesInAssets) = getOperatorAssets(asset_, operator_);
        bytes32 assetLiabilitiesKey = _getAssetLiabilitiesKey(asset_, operator_);
        uint256 operatorLiabilities = _assetLiabilities[assetLiabilitiesKey];
        uint256 borrowedAmount = _borrowedAmounts[assetLiabilitiesKey];

        if (operatorLiabilities > depositedSharesInAssets + borrowedAmount) {
            revert DepositManager_Insolvent(
                address(asset_),
                operatorLiabilities,
                depositedSharesInAssets,
                borrowedAmount
            );
        }
    }
```

If this happens, then functionality that relies on *_validateOperatorSolvency()* won't work anymore. This includes *DepositManager.withdraw(), DepositManager.borrowingWithdraw(), DepositManager.borrowingRepay()* and *DepositManager.borrowingDefault()* functions.

TRST-SR-3 OHM emission depends on market conditions

The OHM emission mechanism implemented by *EmissionManager*, and the bond and convertible deposits auctioneers, does not guarantee a fixed OHM emission schedule. Instead, effective OHM emission depends on market factors, and the mechanism aims to achieve a certain OHM emission rate over the long run and on average. Furthermore, there is no guarantee that the Olympus treasury is compensated according to the market value of the sold convertible deposits.