yAudit

November 26, 2025

Prepared for
Callisto

Audited by
Panda
Adriro

Callisto Vault

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

The Callisto protocol is purpose-built for the Olympus ecosystem, enabling advanced use-cases for OHM holders. It directly interacts with the Olympus contracts and ecosystem tokens OHM, gOHM, and utilizes Olympus Cooler Loans V2 (MonoCooler).

1.2 Audit Scope

This audit covers four smart contracts totaling approximately 700 lines of code across four and a half days of review.

```
src/
├── policies/
│   └── CallistoVault.sol
└── external/
    ├── VaultStrategy.sol
    ├── DebtTokenMigrator.sol
    └── ConverterToWadDebt.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	4
Medium	3
Low	4
Informational	1

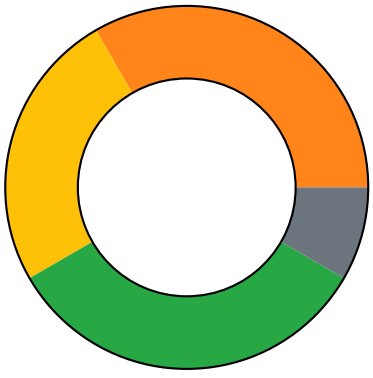

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The Callisto Vault protocol presents a well-architected solution, though the audit revealed several high-severity vulnerabilities related to accounting consistency and edge case handling that the development team promptly addressed. Given the complexity of the architecture and

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: severity classification

its integrations, along with the nature of discovered issues, additional comprehensive testing and security review are strongly recommended before production deployment.

2 Audit Overview

2.1 Project Information

Protocol Name: Callisto
Repository: https://github.com/CallistoDAO/callisto-core/
Commit URL: e7630e33825b85da21a5b3ddbe63c4a43602ad68

2.2 Audit Team

Panda, Adriro

2.3 Audit Timeline

The audit was conducted from August 22 to 28, 2025.

2.4 Audit Resources

Code repositories and documentation

Category	Mark	Description
Access Control	Good	Proper access management is in place.
Mathematics	Low	Multiple issues have been detected affecting the vault's accountability.
Complexity	Average	The contracts are well-architected with a good level of modularization. However, there is an inherent complexity due to the tight coupling with the Olympus infrastructure.
Libraries	Good	Proper use of established libraries like OpenZeppelin and solady.
Decentralization	Good	The vault requires some privileged actors for its management.
Code Stability	Average	The codebase has been heavily updated to address the reported issues.
Documentation	Excellent	Code is well-documented with extensive NatSpec comments. Additionally, the dev team built a dedicated high-level documentation for the audit.
Monitoring	Good	Events for state-changing functions are in place.
Testing and verification	Average	While the codebase includes multiple tests, the vulnerabilities found hint that more testing is needed.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 OHM-gOHM swap accounting mismatch

Assuming a 1:1 cOHM -> gOHM conversion when using an external swapper leads to accounting inconsistencies and potential undercollateralization.

Technical Details

When using `OHMToGOHMMode.Swap`, the vault converts OHM to gOHM via an external swapper but incorrectly assumes a 1:1 conversion for accounting. The vault mints cOHM 1:1 with deposited OHM but uses the actual `gOHMAmount` received (which may be less due to slippage/fees) as collateral. This creates a mismatch between issued shares and actual backing.

Impact

High - Protocol insolvency risk. There will be more cOHM minted than the actual backing.

Recommendation

When OHM->gOHM conversion requires external swapping, pause the standard deposit function and implement a swap-aware deposit function that:

1. Performs the OHM->gOHM swap first
2. Mints cOHM based on the actual gOHM received
3. Maintains accurate 1:1 backing between cOHM shares and gOHM collateral

Note: A residual risk remains when pending deposits exist at the time of switching to swap mode, as these deposits were accounted for assuming direct 1:1 conversion.

Developer Response

This is an acknowledged risk. We do not expect the Swapping use case to be used, so this is added only as a precaution for some truly unexpected circumstances. Even if that is activated, it would most likely only affect the last-to-redeem user, and we expect that to be the protocol itself.

2.6.2 Emergency redeem should account for pending reimbursements

The total invested amount in the strategy should consider pending reimbursements to calculate the owed assets per cOHM.

Technical Details

The implementation of `emergencyRedeem()` uses the assets held by the strategy to calculate the redemption amount.

```
487:     function emergencyRedeem(uint256 shares)
488:         external
489:         override
490:         whenWithdrawalNotPaused
491:         nonzeroValue(shares)
492:         returns (uint256 debtAmount)
493:     {
494:         uint256 debtTokensDeposited = STRATEGY.totalAssetsInvested();
495:         if (_isVaultPositionLiquidated(debtTokensDeposited)) {
496:             /* In an emergency where the vault's position has been liquidated in Olympus Cooler V2 and there are
497:              * funds in the strategy.
498:              */
499:             // Amount to redeem = cOHM amount * Total funds in the strategy / Total cOHM.
500:             debtAmount = Math.mulDiv(shares, debtTokensDeposited, totalSupply());
501:             _burn(msg.sender, shares);
502:             STRATEGY.divest(debtAmount, msg.sender);
503:             emit EmergencyRedeemed(msg.sender, debtAmount);
504:         }
505:         return debtAmount;
506:     }
```

The assets invested in the strategy should also cover the pending reimbursements. If everything is allocated to shareholders, eventually this will lead to insolvent claims.

Impact

High. The issue can cause reimbursement insolvency.

Recommendation

Subtract the

```
+   uint256 availableAssets = debtTokensDeposited - _debtConverterFromWad.convertToDebtTokenAmount(totalReimbursementClaim);
+   debtAmount = Math.mulDiv(shares, availableAssets, totalSupply());
-   debtAmount = Math.mulDiv(shares, debtTokensDeposited, totalSupply());
```

Developer Response

Fixed in commit a080d5a.

2.6.3 Incorrect handling of PSM liquidity during migrations

The migration contract incorrectly assumes a direct relation between the old yield vault shares and the new yield vault.

Technical Details

The implementation of `migrateDebtToken()` scales the current amount of shares supplied by LP to the PSM by simply adjusting by the decimal difference of the underlying tokens.

```
190:         // Gets the total assets supplied in the PSM to convert this value using new decimals.
191:         uint256 suppliedInPSM = psm_.suppliedByLP();
192:
193:         /* Converts amounts to new decimals if necessary.
194:          *
195:          * Warning. When migrating from a debt token with higher decimals to one with lower decimals,
196:          * the operation involves division with upward rounding, ensuring that the Callisto vault has sufficient
197:          * tokens to fully repay its debt in Olympus Cooler Loans V2.
198:          */
199:         IERC20Metadata newDebtToken_ = newDebtToken;
200:         uint8 fromDecimals = IERC20Metadata(address(vault.debtToken())).decimals();
201:         uint8 toDecimals = newDebtToken_.decimals();
202:         uint256 debtTokenAmountConverted;
203:         uint256 suppliedInPSMConverted;
204:         if (fromDecimals > toDecimals) {
205:             uint256 precisionDiff = 10 ** uint256(fromDecimals - toDecimals);
206:             debtTokenAmountConverted = debtTokenAmount.ceilDiv(precisionDiff);
207:             suppliedInPSMConverted = suppliedInPSM.ceilDiv(precisionDiff);
208:         } else if (fromDecimals < toDecimals) {
209:             uint256 precisionDiff = 10 ** uint256(toDecimals - fromDecimals);
210:             debtTokenAmountConverted = debtTokenAmount * precisionDiff;
211:             suppliedInPSMConverted = suppliedInPSM * precisionDiff;
212:         } else {
213:             debtTokenAmountConverted = debtTokenAmount;
214:             suppliedInPSMConverted = suppliedInPSM;
215:         }
```

Eventually, this might be incorrect because:

1. It assumes that both yield vaults have the same number of decimals as their underlying token (so that a difference in asset decimals directly translates to their yield vaults).
2. Share value may have a different relation. Scaling aside, shares may not represent the same amount of underlying.

Impact

High. PSM accounting might break after a migration, causing cascading issues related to the value held by the strategy.

Recommendation

Use the result of `newYieldVault_.deposit({ assets: newDebtTokenAmount, receiver: psmAddr })` to determine the new amount for `suppliedByLP`.

Developer Response

Fixed in commit 9704f43.

2.6.4 Liquidated condition can easily be griefed via donations

After the vault has been liquidated, the condition to have a null collateral amount can be broken via donations, potentially leading to bricked withdrawals.

Technical Details

Given liquidations in MonoCooler seize all the collateral, the implementation of `_isVaultPositionLiquidated()` checks if the position has zero collateral.

```
861:     function _isVaultPositionLiquidated(uint256 totalDeposited) private view
returns (bool) {
862:         /* The condition `totalDeposited > 1` is used instead of `!= 0` because, in
 an extremely rare case,
863:          * 1 token unit may remain on the strategy's balance after withdrawing all
OHM deposits.
864:          * When migrating to a debt token with lower decimals, the division rounds
up to ensure
865:          * the Callisto vault receives exactly enough tokens to cover its debt.
866:          * See `DebtTokenMigrator.migrateDebtToken()` for details.
867:          * Any remaining token unit after withdrawing all deposits is considered as
 an empty balance.
868:          * The minimum debt requirement of Olympus Cooler Loans V2 should prevent
passing this condition when
869:          * not liquidated.
870:          */
871:         return totalDeposited > 1 && OLYMPUS_COOLER.accountCollateral(address(this)
) == 0;
872:     }
```

However, the MonoCooler can operate on behalf of other accounts. In particular, it allows donations via the `addCollateral()` functions.
A donation of just one wei would be enough to block emergency withdrawals, while regular withdrawals would also fail due to the vault's insolvency.

Impact

High. The issue can lead to a denial of service in the withdrawal logic, blocking exits from the vault.

Recommendation

Given the difficulty in correctly assessing whether a position has been liquidated, and considering that this should be a rare event, it might be more reasonable to delegate the task to a trusted operator who can toggle the emergency status.

Developer Response

We have added a state variable `collateralGOHM` that we modify each time the vault position in Cooler v2 is increased or decreased. Now we just compare this value with the actual collateral amount in Cooler v2. If the value in Cooler v2 is less than `collateralGOHM`, we assume the position has been liquidated.
Fixed in commit 9704f43.

2.7 Medium Findings

2.7.1 Timelock protection bypass in debt token migration

The `onlyTimelock` modifier can be bypassed by replacing the DebtTokenMigrator contract, rendering timelock protections ineffective.

Technical Details

The DebtTokenMigrator uses the `onlyTimelock` modifier to ensure migration parameters can only be set after a time delay. However, both CallistoVault and VaultStrategy allow setting a new debt token migrator without timelock protection:

```
File: CallistoVault.sol
377:     function setDebtTokenMigrator(address newMigrator) external onlyRole(
CommonRoles.ADMIN) {
378:         if (newMigrator != address(0)) {
379:             require(
380:                 address(DebtTokenMigrator(newMigrator).OLYMPUS_COOLER()) == address
(OLYMPUS_COOLER),
381:                 MismatchedCoolerAddress()
382:             );
383:         }
384:         address oldMigrator = debtTokenMigrator;
385:         debtTokenMigrator = newMigrator;
386:         emit DebtTokenMigratorSet(oldMigrator, newMigrator);
387:     }
```

```
File: VaultStrategy.sol
148:     function setDebtTokenMigrator(address newMigrator) external onlyOwner {
149:         if (newMigrator != address(0)) {
150:             require(
151:                 address(DebtTokenMigrator(newMigrator).OLYMPUS_COOLER())
152:                     == address(CallistoVault(vault).OLYMPUS_COOLER()),
153:                 MismatchedCoolerAddress()
154:             );
155:         }
156:         address oldMigrator = debtTokenMigrator;
157:         debtTokenMigrator = newMigrator;
158:         emit DebtTokenMigratorSet(oldMigrator, newMigrator);
159:     }
```

An attacker with admin privileges can deploy a malicious migrator contract and execute the migration immediately without any delay.

Impact

Medium - Complete bypass of timelock security controls. If the timelock exists to prevent team misbehavior or protect against compromised admin keys, this bypass allows immediate unauthorized debt token migrations, potentially draining protocol funds or disrupting operations.

Recommendation

Apply timelock protection to `setDebtTokenMigrator()` functions in both contracts, or implement additional governance controls to prevent unauthorized migrator replacement.

Developer Response

Fixed in commit.

2.7.2 Potential denial of service in withdrawals

The withdrawal process will be reverted if the debt to be repaid is zero.

Technical Details

The implementation of `_withdraw()` calculates the amount of debt that needs to be repaid to free the required collateral.

```
623:         /* 1. 3. Calculate the debt amount required to repay a debt in `
OLYMPUS_COOLER` to withdraw
624:          * `gOHMAmountToWithdraw`.
625:          */
626:         (uint128 wadDebt, uint256 debtToRepay) = _calcDebtToRepay(
gOHMAmountToWithdraw);
627:
628:         /* 2. Withdraw the required amount of `debtToken` from the strategy to
repay the debt.
629:          *
630:          * If an emergency case where not enough are available in the strategy,
attempting to obtain
631:          * the lacking amount of `debtToken` from the caller, and record a
reimbursement to be claimed by
632:          * the caller using `claimReimbursement` when there are enough funds in the
 strategy.
633:          * If a caller would not like to pay the lacking amount, then,
alternatively, the caller can withdraw
634:          * the part, for which the available funds are sufficient instead of the
entire amount.
635:          *
636:          * Note. This also occurs if the yield generated by the strategy is not
enough to repay the debt in
637:          * Olympus Cooler Loans V2. For example, if the first depositor attempts to
 immediately withdraw
638:          * the entire initial deposit. So, the strategy has not time to accumulate
yield.
639:          */
640:         IERC20 debtToken_ = debtToken;
641:         _withdrawStrategyOrCallerFunds(debtToRepay, debtToken_);
642:
643:         // 3. Repay the debt in Olympus Cooler V2 to withdraw gOHM.
644:         if (debtToRepay != 0) {
645:             // slither-disable-next-line unused-return
646:             debtToken_.approve(address(OLYMPUS_COOLER), debtToRepay);
647:             // slither-disable-next-line unused-return
648:             OLYMPUS_COOLER.repay({ repayAmountInWad: wadDebt, onBehalfOf: address(
this) });
649:         }
```

In the case `debtToRepay` is zero, the implementation avoids calling `MonoCooler.repay()` due to the check on line 644, but always calls `_withdrawStrategyOrCallerFunds()` on line 641.

```
672:     function _withdrawStrategyOrCallerFunds(uint256 debtToRepay, IERC20 debtToken_)
 private {
673:         // Get the total amount available in the strategy.
674:         uint256 strategyBalance = STRATEGY.totalAssetsAvailable();
675:
676:         // If the strategy covers the entire debt, withdraw the required amount of
`debtToken`.
677:         if (debtToRepay <= strategyBalance) {
678:             STRATEGY.divest(debtToRepay, address(this));
679:             return;
680:         }
```

Note that when `debtToRepay` is zero, the call to `divest()` on line 678 is also executed, which would conflict with the `nonzeroValue` validation and cause a revert in the flow path.

Impact

Medium. Withdrawals may fail in cases where the vault is not max-borrowed.

Recommendation

Move the call to `_withdrawStrategyOrCallerFunds()` inside the `if` of line 644.

Developer Response

Fixed in commit.

2.7.3 Improper WAD scaling

There are instances in the implementation where amounts are not correctly scaled to the debt token domain.

Technical Details

In `_handleDepositsToStrategy()`, the implementation max borrows from the Cooler and sends those assets to the vault strategy.

```
967:         // Returns `debtToken` amount (USDS) and STRATEGY owns them
968:         uint256 debtTokenAmount = OLYMPUS_COOLER.borrow({
969:             borrowAmountInWad: type(uint128).max, // Borrow up to `_globalStateRW().
maxOriginationLtv` of Cooler V2.
970:             onBehalfOf: address(this),
971:             recipient: address(STRATEGY)
972:         });
973:
974:         // 3. Trigger STRATEGY to invest borrowed `debtToken`.
975:         STRATEGY.invest(debtTokenAmount);
```

The `debtTokenAmount` is interpreted as the amount in the scale of the debt token (i.e., USDS). However, technically, this is returned in the WAD scale, as hinted by the interface:

```
    function borrow(uint128 borrowAmountInWad, address onBehalfOf, address recipient)
        external
        returns (uint128 amountBorrowedInWad);
```

The vault tracks pending reimbursements independently of the debt token by converting them to a WAD scale.

```
685:         uint256 reimbursementClaim = debtConverterToWad.toWad(callerContribution);
686:         reimbursementClaims[msg.sender] += reimbursementClaim;
687:         emit ReimbursementClaimAdded(msg.sender, reimbursementClaim,
callerContribution);
688:         totalReimbursementClaim += reimbursementClaim;
```

However, in `totalProfit()`, the `totalReimbursementClaim` is directly used with `STRATEGY.totalAssetsInvested()`, which is expressed in terms of the debt token.

```
824:     function totalProfit() public view override returns (uint256) {
825:         /* Total profit = Strategy funds - Vault's debt to Olympus Cooler V2 -
Total reimbursment to users.
826:          * If the vault is liquidated, profit is 0.
827:          */
828:         uint256 totalDeposited = STRATEGY.totalAssetsInvested();
829:         uint256 totalReimbursement = totalReimbursementClaim;
830:         if (totalDeposited < totalReimbursement) return 0;
831:         if (_isVaultPositionLiquidated(totalDeposited)) return 0;
832:
833:         (, uint256 debt) = _debtConverterFromWad.convertToDebtTokenAmount(
OLYMPUS_COOLER.accountDebt(address(this)));
834:         unchecked {
835:             totalDeposited -= totalReimbursement;
836:         }
837:         return totalDeposited < debt ? 0 : FixedPointMathLib.rawSub(totalDeposited,
 debt);
838:     }
```

Impact

Medium. Accounting might break if the debt token is migrated to a scale that differs from WAD.

Recommendation

```
-       uint256 debtTokenAmount = OLYMPUS_COOLER.borrow({
+       uint256 amountBorrowedInWad = OLYMPUS_COOLER.borrow({
            borrowAmountInWad: type(uint128).max, // Borrow up to `
_globalStateRW().maxOriginationLtv` of Cooler V2.
            onBehalfOf: address(this),
            recipient: address(STRATEGY)
        });
+       uint256 debtTokenAmount = _debtConverterFromWad.
convertToDebtTokenAmount(amountBorrowedInWad);
```

```
-   uint256 totalReimbursement = totalReimbursementClaim;
+   uint256 totalReimbursement = _debtConverterFromWad.convertToDebtTokenAmount
(totalReimbursementClaim);
```

Add tests that use different decimals.

Developer Response

Fixed as part of our big cleanup commit.

2.8 Low Findings

2.8.1 Max withdrawal from the yield vault might affect debt token migration

A limit on withdrawals could cause a partial debt token migration.

Technical Details

The `migrateDebtToken()` function in the DebtTokenMigrator contract uses `ERC4626::maxWithdraw()` to determine the amount of debt tokens that are expected to be migrated.

```
188:         uint256 debtTokenAmount = IERC4626(strategy.yieldVault()).maxWithdraw(
psmAddr);
```

A limit or restriction in the yield vault could result in a lower amount being returned than actually held, potentially leading to a partial debt token migration.

Impact

Low.

Recommendation

While this doesn't affect the current sUSDS vault, consider using `redeem()` with the share balance to draw assets from the vault.

Developer Response

Fixed in PR#12.

2.8.2 Remove approval to the previous yield vault

The `migrateAsset()` function grants an infinite approval to the new yield vault, but never revokes the allowance to the previous vault.

Technical Details

VaultStrategy.sol#L217

Impact

Low.

Recommendation

Revoke the approval to the current `yieldVault` before replacing it.

Developer Response

Fixed and covered by tests in PR#12.

2.8.3 Use SafeERC20 for debt token

While USDS is standard, the debt token might be migrated to a non-standard implementation.

Technical Details

- VaultStrategy.sol#L130
- VaultStrategy.sol#L171
- VaultStrategy.sol#L229
- DebtTokenMigrator.sol#L236
- CallistoVault.sol#L206
- CallistoVault.sol#L304
- CallistoVault.sol#L646
- CallistoVault.sol#L746
- CallistoVault.sol#L1000
- CallistoVault.sol#L1007

Impact

Low.

Recommendation

Use `forceApprove()` to handle approvals.

Developer Response

Fixed in PR#13.

2.8.4 `maxDeposit()` and `maxWithdraw()` should account for pause limits

According to the ERC-4626 standard, both of these functions should account for limits and return zero if disabled.

Technical Details

CallistoVault.sol#L119-L121.

Impact

Low.

Recommendation

Override `maxDeposit()` and `maxWithdraw()` and return zero if the functionality is paused.

Developer Response

Fixed in this commit.

2.9 Gas Savings Findings

2.9.1 Unnecessary ERC20 approvals

Technical Details

`VaultStrategy::invest()`: the yield vault has been already given an infinity approval in the constructor or when migrating to a new one.
`CallistoVault::constructor()`: debt tokens are pushed to the strategy, the strategy doesn't pull from the vault.
`CallistoVault::withdraw()`: the OlympusStaking contract burns GOHM from the caller, it doesn't pull tokens.

Impact

Gas savings.

Recommendation

Remove the highlighted approvals.

Developer Response

The first approval was removed as part of commit
The second remove in commit
`burn()` requires approval, and the test will fail if you try to remove it. We actually now have full control in the constructor in the latest code.

```
GOHM.approve(address(OLYMPUS_STAKING), type(uint256).max);
```

2.9.2 Unnecessary in-memory variable declaration

Some memory variables are declared, but the code can be written without using them, reducing gas usage.

Technical Details

The following block:

```
        address oldMigrator = debtTokenMigrator;
        debtTokenMigrator = newMigrator;
        emit DebtTokenMigratorSet(oldMigrator, newMigrator);
```

Can be replaced by:

```
        emit DebtTokenMigratorSet(debtTokenMigrator, newMigrator);
        debtTokenMigrator = newMigrator;
```

It is found on: VaultStrategy.sol#L156-L158 and CallistoVault.sol#L384-L386
The following line can be removed in favor of using the PSM immutable variable.

```
File: VaultStrategy.sol
199:         CallistoPSM psm = PSM;
```

VaultStrategy.sol#L199-L199

Impact

Gas savings.

Recommendation

Update the code to reduce gas usage.

Developer Response

These functions were removed in the latest version.

2.10 Informational Findings

2.10.1 Deposit pauses don't affect queued OHM

While new deposits are forbidden, pending OHM waiting to be staked is not affected by pauses.

Technical Details

CallistoVault.sol#L119.

Impact

Informational.

Recommendation

Consider if the deposit pause should also halt gOHM staking and/or swaps.

Developer Response

Pending OHM can always be withdrawn when cOHM is redeemed, so we are fine with that.