Electisec Olympus Cooler V2 Review

Review Resources:

- Documentation
- Protocol diagram flows

Auditors:

- Panda
- Adriro

Table of Contents

Review Summary

Olympus Cooler V2

Cooler V2 lending system allows users to borrow USDS against gOHM collateral. Key differences from V1:

1. Perpetual positions instead of fixed-term loans
2. Single unified position per user instead of separate loan entries
3. Governance-controlled LTV growth through a drip system
4. Built to be integrated into other protocols

The contracts of the Olympus repository were reviewed over 8 days. Two auditors performed the code review between February 18th and February 27th, 2025. The repository was under active development during the review, but the review was limited to the latest commit 6b702f06d5bf90e8729d8b8b7baaeaffd54383fd of the Olympus repository.

Scope

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

```
src
├── external
│   └── cooler
│       ├── DelegateEscrow.sol
│       └── DelegateEscrowFactory.sol
├── libraries
│   ├── CompoundedInterest.sol
│   └── SafeCast.sol
├── modules
```

```
│   └── DLGTE
│       ├── DLGTE.v1.sol
│       ├── IDLGTE.v1.sol
│       └── OlympusGovDelegation.sol
└── policies
    ├── cooler
    │   ├── CoolerLtvOracle.sol
    │   ├── CoolerTreasuryBorrower.sol
    │   └── MonoCooler.sol
    └── utils
        ├── PolicyAdmin.sol
        ├── PolicyEnabler.sol
        └── RoleDefinitions.sol
```

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

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

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

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The protocol implements proper role-based access control for administrative functions. Only one minor issue was found with authorization deadline checks.
Mathematics	Good	The interest calculations and liquidation mechanisms appear to be implemented correctly, with only a minor edge case identified related to small amounts of burns during liquidations.
Complexity	Good	The codebase demonstrates appropriate complexity for its functionality, with a clear separation of concerns between components.
Libraries	Good	The protocol uses established libraries well and includes custom utility libraries like CompoundedInterest and SafeCast for specific needs.
Decentralization	Good	The system includes proper governance controls while balancing admin functions and user autonomy.
Code stability	Good	The codebase appears stable, with minimal issues identified during the audit period.
Documentation	Good	The code includes appropriate documentation and external resources were provided to help understand the system architecture.
Monitoring	Good	The protocol implements proper event emission for key state changes, facilitating off-chain monitoring.
Testing and verification	Good	The codebase was well-tested before the audit.

Findings Explanation

Findings are broken down into sections by their respective impact:

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

- Gas savings
    - Findings that can improve the gas efficiency of the contracts.
- Informational
    - Findings including recommendations and best practices.

Critical Findings

None

High Findings

None

Medium Findings

1. Medium - Front-running liquidation with delegation can revert liquidation

Liquidation can be prevented by frontrunning the liquidation transaction with delegation changes.

Technical Details

The `_undelegateForLiquidation` function in `MonoCooler.sol` is vulnerable to a frontrunning attack that could prevent legitimate liquidations. The issue arises when a user-facing liquidation is frontrunning the liquidation transaction by modifying their delegations (e.g., changing delegation addresses or amounts).

Let's say we have a user with the following state:

- Total collateral: 100 gOHM
- Delegated to Bob: 80 gOHM

- Undelegated: 20 gOHM

The user's position becomes liquidatable. A liquidator submits a transaction to batchLiquidate with a delegation request to undelegate 80 gOHM from Bob.

However, the user sees this pending transaction and frontruns it with their own transaction that:

- Undelegates 70 gOHM from Bob
- Delegates 70 gOHM to Alice Now, when the liquidator's transaction executes:

The request to undelegate 80 gOHM from Bob will fail because there are only 10 gOHM delegated to Bob now. the liquidation will fail.

Impact

Medium.

Recommendation

When a position is liquidable, allow only the owner to add funds to prevent liquidation without permitting changes to delegations.

Developer Response

Fixed #53

Low Findings

1. Low - Add an upper limit

Technical Details

The `setInterestRateWad()` function in MonoCooler.sol allows an admin to set any interest rate without an upper bound. While this requires admin privileges, having no maximum cap creates unnecessary risk if the admin role is compromised or if a mistake is made.

```
File: MonoCooler.sol
651:     function setInterestRateWad(uint96 newInterestRate) external override
```

```
onlyAdminRole {
652:         // Force an update of state on the old rate first.
653:         _globalStateRW();
654:
655:         emit InterestRateSet(newInterestRate);
656:         interestRateWad = newInterestRate;
657:     }
```

[MonoCooler.sol#L651-L657]()

Impact

Low.

Recommendation

Make sure there is an upper limit to the interest rate.

Developer Response

Fixed [e42572]()

2. Low - Potential denial of service during liquidations

Liquidations may revert for small amounts of burned gOHM.

Technical Details

The [`batchLiquidate()`]() function burns the seized gOHM collateral tokens.

```
584:         // burn the gOHM collateral and update the total state.
585:         if (totalCollateralClaimed > 0) {
586:             // Unstake and burn gOHM holdings.
587:             uint128 gOhmToBurn = totalCollateralClaimed -
totalLiquidationIncentive;
588:
589:             if (gOhmToBurn > 0) {
590:                 _COLLATERAL_TOKEN.safeApprove(address(_STAKING), gOhmToBurn);
591:                 MINTR.burnOhm(
592:                     address(this),
593:                     _STAKING.unstake(address(this), gOhmToBurn, false, false)
```

```
                );
            }

            totalCollateral -= totalCollateralClaimed;
        }
```

Before burning the tokens, the implementation must first unstake the gOHM into OHM. The relation between gOHM and OHM is given by the `balanceFrom()` function:

```
/**
    @notice converts gOHM amount to OHM
    @param _amount uint
    @return uint
  */
function balanceFrom(uint256 _amount) public view override returns (uint256) {
    return _amount.mul(index()).div(10**decimals());
}
```

Given this relation is not 1:1, a non-zero amount of gOHM could be unstaked as a zero OHM amount. If this happens, the call to `burnOhm()` would fail, as it expects a non-zero amount of tokens.

```
    function burnOhm(
        address from_,
        uint256 amount_
    ) external override permissioned onlyWhileActive {
        if (amount_ == 0) revert MINTR_ZeroAmount();

        ohm.burnFrom(from_, amount_);

        emit Burn(msg.sender, from_, amount_);
    }
```

Impact

Low. The magnitude of the issue is lowered due to the small amount of gOHM

Recommendation

Ensure the resulting amount from unstaking is greater than zero.

```
    if (gOhmToBurn > 0) {
        _COLLATERAL_TOKEN.safeApprove(address(_STAKING), gOhmToBurn);
+       uint256 ohmAmount = _STAKING.unstake(address(this), gOhmToBurn, false,
false);
+       if (ohmAmount > 0) {
          MINTR.burnOhm(
              address(this),
-             _STAKING.unstake(address(this), gOhmToBurn, false, false)
+             ohmAmount
          );
+       }
    }
```

Developer Response

Fixed 87ac82.

Gas Saving Findings

1. Gas - Allowance to Staking contract is not needed

The OlympusStaking contract can burn gOHM tokens from the caller without the need for approval.

Technical Details

MonoCooler.sol#L589-L595

```
589:             if (gOhmToBurn > 0) {
590:                 _COLLATERAL_TOKEN.safeApprove(address(_STAKING), gOhmToBurn);
591:                 MINTR.burnOhm(
592:                     address(this),
593:                     _STAKING.unstake(address(this), gOhmToBurn, false, false)
594:                 );
595:             }
```

Impact

Gas savings.

Recommendation

Remove the approval on line 590.

Developer Response

Fixed 87ac82

2. Gas - Use unchecked math if there is no overflow risk

There are math operations that can use unchecked arithmetic for gas savings.

Technical Details

- MonoCooler.sol#L339
- MonoCooler.sol#L351
- MonoCooler.sol#L405
- MonoCooler.sol#L466
- CoolerTreasuryBorrower.sol#L113
- DelegateEscrow.sol#L79

Impact

Gas savings.

Recommendation

Use unchecked block if there is no overflow or underflow risk for gas savings.

Developer Response

Fixed 197564ea.

Informational Findings

1. Informational - Authorization check incorrectly handles authorization deadline

Technical Details

In the `MonoCooler` contract, the `isSenderAuthorized` function contains a logic error in how it handles authorization deadlines. The current implementation prevents authorized actions at the deadline timestamp when they should still be allowed.

```
File: MonoCooler.sol
    function isSenderAuthorized(address sender, address onBehalfOf) public view returns (bool) {
        return sender == onBehalfOf || block.timestamp < authorizations[onBehalfOf][sender];
    }
```

Impact

Informational.

Recommendation

Change the comparison operator from < to <= to allow operations at the deadline timestamp.

```
function isSenderAuthorized(address sender, address onBehalfOf) public view returns (bool) {
    return sender == onBehalfOf || block.timestamp <= authorizations[onBehalfOf][sender];
}
```

Developer Response

Fixed 65f8748

2. Informational - PolicyEnabler features are not used in CoolerLtvOracle.sol

The contract inherits from PolicyEnabler but does not use its features to prevent execution when the state is disabled.

Technical Details

CoolerLtvOracle.sol#L21

```
21: contract CoolerLtvOracle is ICoolerLtvOracle, Policy, PolicyEnabler {
```

Impact

Informational.

Recommendation

Double-check if anything should be paused while disabled, or replace PolicyEnabler with PolicyAdmin to remove the enable/disable functionality.

Developer Response

Fixed cdaae3

Final remarks

Overall, the Olympus Cooler V2 protocol demonstrates a well-structured and secure implementation. Our review identified no critical or high-severity issues, with only one medium-severity finding related to potential front-running of liquidations, two low-severity issues, and a few gas optimizations and informational notes. The codebase shows good practices in access control, mathematical operations, and architecture design. We recommend the Olympus team address the identified findings before mainnet deployment, mainly focusing on mitigating the liquidation front-running vulnerability to ensure the protocol's liquidation mechanism functions as intended under all circumstances.