Electisec Olympus Cooler - Migrator and Composite Review

Review Resources:

None beyond the code repositories.

Auditors:

- Panda
- Adriro

Table of Contents

Review Summary

Olympus Cooler - Migrator and Composite

The review covers the following three pull requests in the Olympus V3 repository:

1. Cooler V2: Composite: `12d5f2ab2a187aafd6041c2b12b291b281506e31`
2. Cooler V2: Migrator: `f719a2996af687b218c90a3b569b68d98089b97c`
3. Cooler V2: Migrator improvements: `e8f6803c7e14fbc26ea99aa3d45c9cc067f37bf6`

Two auditors reviewed these pull requests between March 7 and March 12, 2025. While the repository was under active development during the review period, the review focused specifically on the changes introduced in these three pull requests.

Scope

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

Composites

```
src/policies/cooler
|── CoolerComposites.sol
```

Migrator

```
src/policies/cooler/
|── CoolerV2Migrator.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.

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

1. Critical - Attacker can migrate bad debt to victims

Since old Cooler loans are peer-to-peer, an attacker can intentionally create a loan with bad debt and donate it to a victim who has authorized the migrator contract.

Technical Details

Existing loans in old Cooler implementations can be migrated to a new owner in the Cooler V2 contract.

Existing loans would normally be healthy, and transferring those to the new Cooler should maintain that property as long as the new LTV is at least the old LTV.

However, since old Cooler loans are peer-to-peer and permissionless, an attacker can intentionally create a loan with bad debt as they can choose their terms and then donate that loan to a new owner who has previously given authorization to the migrator contract.

This way, the attacker can take advantage of any existing collateral belonging to the victim and the potential margin for their debt.

Proof of Concept

Place the following test in **`CoolerV2Migrator.t.sol`**.

```
function test_Migrate_BadDebt() public {
    (address user, uint256 userPk) = makeAddrAndKey("user");
    (address attacker, uint256 attackerPk) = makeAddrAndKey("attacker");

    uint256 attackerCollateral = 1;
    uint256 attackerDebt = 1_000e18;

    uint128 victimCollateral = 10e18;

    gohm.mint(attacker, attackerCollateral);
    dai.mint(attacker, attackerDebt);

    gohm.mint(user, victimCollateral);

    // Simulate user has migrated to new cooler so there is a dangling
```

```
authorization. User has enough margin to absorb debt.
    vm.startPrank(user);

    gohm.approve(address(cooler), type(uint256).max);

    cooler.addCollateral(victimCollateral, address(user), delegationRequests);

    cooler.setAuthorization(address(migrator), type(uint96).max);

    vm.stopPrank();

    vm.startPrank(attacker);

    Cooler oldCooler = Cooler(coolerFactory.generateCooler(gohm, dai));

    gohm.approve(address(oldCooler), type(uint256).max);
    gohm.approve(address(migrator), type(uint256).max);
    dai.approve(address(oldCooler), type(uint256).max);

    // Attacker loans to himself 1_000e18 DAI with 1 wei of gOHM
    oldCooler.requestLoan(attackerDebt, 0, 1e39, 1 days);
    oldCooler.clearRequest(0, attacker, false);

    address[] memory coolers = new address[](1);
    coolers[0] = address(oldCooler);
    address[] memory houses = new address[](1);
    houses[0] = address(clearinghouseDai);

    IMonoCooler.Authorization memory auth;
    IMonoCooler.Signature memory sig;

    // Attacker migrates debt to user
    migrator.consolidate(
        coolers,
        houses,
        user,
        false,
        auth,
        sig,
        delegationRequests
```

```
    );

    vm.stopPrank();

    // Attacker has doubled their DAI
    assertEq(dai.balanceOf(attacker), 2 * attackerDebt);
}
```

Impact

Critical. A malicious actor can transfer bad debt to users who have previously authorized the migrator contract to operate on their behalf.

Recommendation

When migrating loans, ensure the lender in the old Cooler is trusted and that the LTV is lower than the LTV in the new Cooler V2. Additionally, given the findings related to the dangling authorization, it is recommended that this authorization be revoked once the migrator uses it.

Developer Response

Fixed by adding a lender check in abf43d9.

High Findings

1. High - Attacker can control delegations using CoolerV2Migrator

An attacker can abuse the migration process to tamper with other users' delegations.

Technical Details

The migration process allows users to set a new owner when loans are migrated to the Cooler V2.

Since the process involves applying delegations and borrowing USDS, the new owner of the position must authorize the migrator contract to operate on their behalf, either by bundling the required signature in the migration or by calling `setAuthorization()` before starting the process.

A malicious actor can abuse this process to control delegations of users who have already given this authorization. By creating a dummy loan with a small amount of collateral and debt, they can set the new owner as the target and provide an arbitrary array of delegation requests to modify any existing gOHM the victim has already deposited.

Impact

High. Attackers can control delegations for users who authorized the migrator contract.

Recommendation

When migrating a loan, ensure that the delegation requests don't contain an undelegation and that the sum of each amount doesn't exceed the total collateral being migrated.

Developer Response

Fixed in PR#57.

Medium Findings

None

Low Findings

1. Low - Missing USDS approval to Sky Migrator

The CoolerV2Migrator.sol fails to grant the USDS allowance to the Sky migrator contract to convert the borrowed USDS into DAI to repay the flashloan.

Technical Details

CoolerV2Migrator.sol#L412-L414

```
412:         // Convert the USDS to DAI
413:         uint256 usdsBalance = _USDS.balanceOf(address(this));
414:         if (usdsBalance > 0) MIGRATOR.usdsToDai(address(this), usdsBalance);
```

Impact

Low. The issue leads to a denial of service in the migration process, but no funds are lost.

Recommendation

Set the USDS allowance to the `MIGRATOR` contract.

```
// Convert the USDS to DAI
uint256 usdsBalance = _USDS.balanceOf(address(this));
if (usdsBalance > 0) {
  _USDS.safeApprove(address(MIGRATOR), usdsBalance);
  MIGRATOR.usdsToDai(address(this), usdsBalance);
}
```

Developer Response

Fixed in 6520d11.

2. Low - Debt token leftovers in Composite when repaying the entire debt

Leftovers are not refunded when the repayment amount exceeds the debt of the position.

Technical Details

MonoCooler's `repay()` function caps the repay amount to the account's current debt.

```
470:         // Cap the amount to be repaid to the current debt as of this block
471:         if (repayAmount < latestDebt) {
472:             amountRepaid = repayAmount;
473:
474:             // Ensure the minimum debt amounts are still maintained
475:             unchecked {
476:                 aState.debtCheckpoint = _accountDebtCheckpoint = latestDebt -
amountRepaid;
477:             }
478:             if (_accountDebtCheckpoint < _MIN_DEBT_REQUIRED) {
479:                 revert MinDebtNotMet(_MIN_DEBT_REQUIRED,
_accountDebtCheckpoint);
480:             }
481:         } else {
482:             amountRepaid = latestDebt;
```

```
483:             aState.debtCheckpoint = 0;
484:         }
```

If the user specifies a repayment amount with some extra margin to consider any potential interest that could be generated until the transaction gets executed, the implementation of `repayAndRemoveCollateral()` will pull the entire amount of debt tokens but fail to refund any excess that could be present after the operation.

```
61:         _DEBT_TOKEN.safeTransferFrom(msg.sender, address(this), repayAmount);
62:         COOLER.repay(repayAmount, msg.sender);
63:         COOLER.withdrawCollateral(collateralAmount, msg.sender, msg.sender,
delegationRequests);
```

Impact

Low. Potential leftovers are not refunded to the caller and will remain locked in the contract.

Recommendation

In `repayAndRemoveCollateral()`, cap the repaid amount to the account's current debt. This will ensure no leftover is present after the operation and will help improve the contract's UX since a user can easily repay the entire debt by specifying `type(uint256).max`.

Developer Response

Fixed in dc13881.

Gas Saving Findings

1. Gas - Optimize gas usage when a signature is not required

In both addCollateralAndBorrow and repayAndRemoveCollateral functions, once the signature authorization has been set, it will remain a parameter, increasing the calldata unnecessarily.

Technical Details

```
if (authorization.account != address(0)) {
    COOLER.setAuthorizationWithSig(authorization, signature);
```

```
}
```

CoolerComposites.sol#L41-L41

CoolerComposites.sol#L58-L58

Impact

Gas savings.

Recommendation

The function can be split so that once the signature is unnecessary, it can be called without it.

```
function addCollateralAndBorrow(
        IMonoCooler.Authorization memory authorization,
        IMonoCooler.Signature calldata signature,
        uint128 collateralAmount,
        uint128 borrowAmount,
        IDLGTEv1.DelegationRequest[] calldata delegationRequests
    ) external {
            COOLER.setAuthorizationWithSig(authorization, signature);
            addCollateralAndBorrow(collateralAmount, borrowAmount,
delegationRequests);
    }

function addCollateralAndBorrow(
        IMonoCooler.Signature calldata signature,
        uint128 collateralAmount,
        uint128 borrowAmount,
        IDLGTEv1.DelegationRequest[] calldata delegationRequests
    ) public {
        _COLLATERAL_TOKEN.safeTransferFrom(msg.sender, address(this),
collateralAmount);
        COOLER.addCollateral(collateralAmount, msg.sender, delegationRequests);
        COOLER.borrow(borrowAmount, msg.sender, msg.sender);
    }
```

Developer Response

Acknowledged.

2. Gas - Cooler addresses can be sorted off-chain

Addresses can be sorted off-chain to prevent easier duplication checks in the array of Coolers.

Technical Details

To check for duplicates, the array of Cooler data is traversed again for each Cooler.

```
257:             // Check that the Cooler is not already in the array
258:             for (uint256 j; j < coolerData.length; j++) {
259:                 if (address(coolerData[j].cooler) == coolers_[i]) revert
Params_DuplicateCooler();
260:             }
```

If Cooler addresses are sorted off-chain, the implementation can quickly check for duplicates by requiring that the current address is greater than the previous one.

Impact

Gas savings.

Recommendation

Check that Cooler addresses are sorted.

```
  if (i > 0 && uint160(coolers_[i-1]) >= uint160(coolers_[i])) revert
Params_DuplicateCooler();
```

Developer Response

Acknowledged.

Informational Findings

1. Informational - Log unexpected funds return

Technical Details

```
File: CoolerV2Migrator.sol
335:         // This shouldn't happen, but transfer any leftover funds back to the
sender
336:         uint256 usdsBalanceAfter = _USDS.balanceOf(address(this));
337:         if (usdsBalanceAfter > 0) {
338:             _USDS.safeTransfer(msg.sender, usdsBalanceAfter);
339:         }
340:         uint256 daiBalanceAfter = _DAI.balanceOf(address(this));
341:         if (daiBalanceAfter > 0) {
342:             _DAI.safeTransfer(msg.sender, daiBalanceAfter);
343:         }
```

It shouldn't happen, but if it does, it might be good to have an event to determine quickly when a transfer of leftover funds happens.

Impact

Informational.

Recommendation

Add an event to monitor the case.

Developer Response

Fixed in 6471738.

2. Informational - Missing check for collateral token

The migrator contract never checks if the collateral token is gOHM for existing Cooler vaults.

Technical Details

The CoolerV2Migrator.sol implementation validates that the debt token is either DAI or USDS but never checks if the collateral token is gOHM.

Impact

Informational.

Recommendation

For each Cooler, validate that the actual collateral token is gOHM.

Developer Response

This isn't an issue as the gOHM was being transferred from the cooler owner after repayment. If the Cooler did not have gOHM collateral, then the transfer would not succeed.

Final remarks

The security audit revealed several important issues, including a critical vulnerability allowing bad debt migration to victims, a high risk of delegation control abuse, and low-severity issues with approvals and token refunds. The team addressed the major concerns through prompt fixes.