CANTINA

Seamless

Security Review

Cantina Managed review by:

Denis Miličević, Lead Security Researcher
Om parikh, Security Researcher

October 1, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Seamless turns leveraged DeFi strategies into simple ERC-20 tokens - easy to access, easy to use.

From Sep 3rd to Sep 8th the Cantina team conducted a review of leverage-tokens on commit hash 063cf5bc. The team identified a total of **9** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	0	0	0
Low Risk	3	3	0
Gas Optimizations	2	1	1
Informational	3	3	0
Total	**9**	**8**	**1**

3 Findings

3.1 High Risk

3.1.1 Unauthorized Transfer of User-Approved ERC20 Tokens via `LeverageRouter`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In `_depositAndRepayMorphoFlashLoan` & `_redeemAndRepayMorphoFlashLoan` internal functions, it allows arbitrary external calls to the any addresses which are passed as `Call[] calldata swapCalls` from `deposit` or `redeem` entrypoint:

```
for (uint256 i = 0; i < params.swapCalls.length; i++) {
    // slither-disable-next-line unused-return
    Address.functionCallWithValue(
        params.swapCalls[i].target, params.swapCalls[i].data, params.swapCalls[i].value
    );
}
```

Since `LeverageRouter` is used for minting and redeeming leverage tokens by end users, It needs to have ERC20 allowance for minting shares and leverage tokens allowance for redeeming. These are approved just in time before actual mint / redeem transaction or may have infinite allowance if given by user.

However, A malicious actor can drain all funds by adding a `transferFrom` call in `swapCalls` input to deposit / redeem function on ERC20 from user who has approved, and `to` address being their controlled address.

Other similar paths exist to extract to steal the approved funds & to take allowance so that future funds could be stolen such as:

- Calling `morpho.flashLoan` from arbitrary call which triggers callback with data which is not sanitized.
- Calling `permit2` to take the infinite token allowance of router where spender is attacker controlled address.

This is especially more exploitable on chains which public mempool such as ethereum mainnet.

Proof of Concept:

```
function test_Alice_steals_bob_approval() public {

    address alice = makeAddr("alice");
    uint bobsCollateral = 123 ether;
    // Execute the deposit
    deal(address(collateralToken), address(this), bobsCollateral);
    collateralToken.approve(address(leverageRouter), bobsCollateral);

    ILeverageRouter.Call[] memory aliceStealCall = new ILeverageRouter.Call[](1);
    aliceStealCall[0] = ILeverageRouter.Call({
        target: address(collateralToken),
        data: abi.encodeWithSelector(IERC20.transferFrom.selector, address(this), address(alice),
        ↪  bobsCollateral),
        value: 0
    });


    // Alice starts with no collateral token
    assertEq(collateralToken.balanceOf(alice), 0);

    _mockLeverageManagerDeposit(0, 0, 0, 0);
    vm.prank(alice);
    leverageRouter.deposit(leverageToken, 0, 0, 0, aliceStealCall);

    // now Alice has Bob's collateral
    assertEq(collateralToken.balanceOf(alice), bobsCollateral);
}
```

Recommendation: There are two main ways of tackling this:

- All externals calls should be fully validated for address and calldata being trusted and whitelisted to ensure there are no risks. Anything which is in path of morpho flashloan callback should check for re-entrancy.

- Use a separate contract to make untrusted externals which doesn't hold any asset, approvals or privilleged roles outside of a single transaction context.

Seamless Protocol: Fixed in PR 279.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 `PricingAdapter` doesn't account for fee adjusted supply, leading to potential overpricing

Severity: Low Risk

Context: PricingAdapter.sol#L26-L38, PricingAdapter.sol#L41-L53

Description: The `PricingAdapter` contract is intended to provide pricing denominations of the leverage tokens in their respective collateral or debt tokens. The logic for calculating this utilizes the leverage tokens `totalSupply()` function, which in cases that a management fee is active, fails to account for the accrued fees on the supply, and thereby underreports the actual supply present, which results in an overpriced output for contracts or off-chain utilities depending on the pricing.

Recommendation: Utilize the `getFeeAdjustedTotalSupply(ILeverageToken token)` function instead to provide a more accurate pricing output.

Seamless Protocol: Fixed in commit b2127b36.

Cantina Managed: Fix verified.

3.2.2 Ceiling rounding in `_getAccruedManagementFee` can yield a much higher effective fee being charged than set

Severity: Low Risk

Context: FeeManager.sol#L259-L260

Description: In almost all cases, with the current logic implemented in `_getAccruedManagementFee`, the set management fee, is basically the minimum fee that would be effectively charged. In most instances, it will be at least fractionally greater, under exceptional circumstances, the effective fee could end up being orders of magnitude higher through the lifetime of the token, with those scenarios being a low supply token resultant from a donation attack which would be further amplified by a low management fee.

Recommendation: Switch the logic to do a floor rounding, which will now make the set fee be the maximal possible fee chargable by the protocol under ideal scenarios. In reality it will be slightly lower, in the worst case of an exceptional scenario, where all users of the contract purposefully abuse the floor rounding by specifically timing all transactions, and any users, including the team have their transactions censored, the effective fee could be cut in half, which is a much lower range than in the prior exceptional cases, which are also more likely to occur, this noted scenario is very unlikely to occur. This change would effectively improve the consistency and accuracy of the management fee.

There will need to be more logic changes, as the `lastManagementFeeAccrualTimestamp[token]` storage variable for the token cannot be updated at all times, but only in the case the fee is actually charged, and management fee is actually set, as we don't want duration for fees increasing while there is no fee set, allowing potentially for a ”fee bomb” to be set all at once. Essentially a check for `managementFee != 0 && feeShares == 0` which early returns and doesn't update the fee accrual timestamp if the condition is met.

Seamless Protocol: Fixed in commit fa248fd6.

Cantina Managed: Fix verified.

3.2.3 `_computeFeesForNetShares` fails to account for additional rounding step in reversal of `_computeFeesForGrossShares`

Severity: Low Risk

Context: FeeManager.sol#L228-L238

Description: The logic in `_computeFeesForNetShares` discounts an additional rounding step that occurs in `_computeFeesForGrossShares`. This leads to additional off-by-one cases between `previewRedeem` and `previewWithdraw` outputs on the `treasuryFee` action data.

An example showcasing this:

```
Preview redeem 10 shares returns:
  8 collateral
  4 debt
  10 shares
  1 token fee share
  1 treasury fee share

 Preview withdraw 8 shares returns:
  8 collateral
  4 debt
  9 shares
  1 token fee share
  0 treasury fee share
```

With the same equivalent inputs they should ideally match, otherwise the lower fee call of withdraw could be detrimental to the treasury.

Recommendation: Account for the additional rounding step by switching the logic to:

```
grossShares = Math.mulDiv(
    Math.mulDiv(
        netShares,
        MAX_BPS,
        (MAX_BPS - tokenActionFeeRate),
        Math.Rounding.Ceil
    ),
    MAX_BPS,
    MAX_BPS - treasuryActionFeeRate,
    Math.Rounding.Ceil
);
```

This solution resolves the above cases mismatch, but there is still some precision loss which results in a mismatch of 1 for other cases between `previewRedeem` and `previewWithdraw`, however, from fuzz tests this logic halves the occurrance those cases compared to the current logic.

Seamless Protocol: Fixed in commit fa248fd6.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Struct in `FeeManager` has variables that could be tightly packed

Severity: Gas Optimization

Context: FeeManager.sol#L50-L53

Description: The `FeeManagerStorage` struct in the `FeeManager` contract has 2 separate mappings pointing to `uint256` types that have code paths leading to storage loading and storing within the same call. Additionally, the types can safely be bounded to smaller `uint` types.

Recommendation: The mappings should be condensed into a single mapping, pointing to a struct that contains the `managementFee` and `lastManagementFeeAccrualTimestamp`. A safe type for `managementFee` is `uint16`, and `uint120` for `lastManagementFeeAccrualTimestamp`, which can save a `SSTORE` or `SLOAD` opcode when these are both accessed.

Seamless Protocol: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 `_getAccruedManagementFee` can be optimized for on-chain callpaths

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The `_getAccruedManagementFee` attempts some operations that are redundant when accessed through `deposit`, `mint`, `redeem`, `withdraw`, which is wasting gas, as the `chargeManagementFee` function will have already run through these operations and appropriately adjust the `totalSupply()` with fees.

Recommendation: Have the function early return 0, skipping unnecessary operations, and by re-ordering a portion, allows the skipping of an additional unnecessary warm storage load, saving at least 100 gas, from:

```
function _getAccruedManagementFee(ILeverageToken token, uint256 totalSupply) internal view returns (uint256) {
    uint256 managementFee = getManagementFee(token);
    uint120 lastManagementFeeAccrualTimestamp = getLastManagementFeeAccrualTimestamp(token);

    uint256 duration = block.timestamp - lastManagementFeeAccrualTimestamp;

    uint256 sharesFee =
        Math.mulDiv(managementFee * totalSupply, duration, MAX_BPS * SECS_PER_YEAR, Math.Rounding.Ceil);
    return sharesFee;
}
```

to:

```
function _getAccruedManagementFee(ILeverageToken token, uint256 totalSupply) internal view returns (uint256) {
    uint120 lastManagementFeeAccrualTimestamp = getLastManagementFeeAccrualTimestamp(token);
    uint256 duration = block.timestamp - lastManagementFeeAccrualTimestamp;

    if (duration == 0)
        return 0;

    uint256 managementFee = getManagementFee(token);

    uint256 sharesFee =
        Math.mulDiv(managementFee * totalSupply, duration, MAX_BPS * SECS_PER_YEAR, Math.Rounding.Ceil);
    return sharesFee;
}
```

Seamless Protocol: Fixed in commit 65fc8893.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Public function with internal variant, should just be set to external

Severity: Informational

Context: LeverageManager.sol#L134

Description: The function `convertCollateralToShares` in the `LeverageManager` contract has its visibility set to public, although internally it is accessed via the internal variant `_convertCollateralToShares` which it calls itself into as well. This is likely an oversight or typo, as all the other similar functions within the contract follow the external/internal pattern already.

Recommendation: Set its visibility specifier to external, to avoid issues arising between internal use of the internal or external facing function.

Seamless Protocol: Fixed in commit db5a0547.

Cantina Managed: Fix verified.

3.4.2 New mints are possible post complete liquidation

Severity: Informational

Context: LeverageManager.sol#L642-L647

Description: one could deposit directly on lending adapter on behalf of lending adapter when `totalSupply !=0 && totalCollateral == 0` so that both becomes non-zero now. this undermines the stated assumption in commnets and whoever deposits can control (and possibly skew) the exchange rate. However, there are no major implications as shares were already worth nothing after liquidation.

Recommendation:

- The comments (and/or any other related docs) should be updated if deemed important to outline this scenario.
- If it should not be possible to mint at all in this case, it should be enforced via some stricter mechanism.

Seamless Protocol: Fixed in commit 57f163f2.

Cantina Managed: Fix verified. Natspec was updated in commit 57f163f2 to reflect the correct behaviour.

3.4.3 `PricingAdapter` should be discouraged for on-chain use

Severity: Informational

Context: PricingAdapter.sol#L56-L73

Description: `getLeverageTokenPriceAdjusted` returns oracle price adjusted value. For example: If oracle = DAI / USD, then `oraclePrice` = `0.99e8` debt token is DAI, `baseAssetDecimals` is 18, then adjusted price can round down to 0 or be extremely imprecise depending on precision of base asset and oracle used.

Recommendation: Consider documenting that it should be only used for offchain batching of calls and not for on-chain callers / integrations as the precision & rounding direction used may not be suitable for all use cases.

Seamless Protocol: Fixed in commit 4144bcbf.

Cantina Managed: Fix verified.