SHERLOCK

Security Review For Seamless Protocol

Collaborative Audit Prepared For: **Seamless Protocol**
Lead Security Expert(s): **0x52**
Kirkeelee

Date Audited: **April 29 - May 8, 2025**
Final Commit: **6c745a1**

Introduction

Leverage tokens are meant to make leverage easy and permissionless. Allowing for permissionless tokenization of a Leverage Position on any DeFi lending protocol, starting with Morpho.

Scope

Repository: seamless-protocol/leverage-tokens

Audited Commit: 23896d613d8509c785bc419851f6137647e8b610

Final Commit: 6c745a1fb2c5cc77df7fd3106f57db1adc947b75

Files:

- src/BeaconProxyFactory.sol
- src/FeeManager.sol
- src/LeverageManager.sol
- src/LeverageToken.sol
- src/lending/MorphoLendingAdapter.sol
- src/lending/MorphoLendingAdapterFactory.sol
- src/periphery/EtherFiLeverageRouter.sol
- src/periphery/LeverageRouter.sol
- src/periphery/LeverageRouterBase.sol
- src/periphery/LeverageRouterMintBase.sol
- src/periphery/SwapAdapter.sol
- src/rebalance/CollateralRatiosRebalanceAdapter.sol
- src/rebalance/DutchAuctionRebalanceAdapter.sol
- src/rebalance/PreLiquidationRebalanceAdapter.sol
- src/rebalance/RebalanceAdapter.sol
- src/types/DataTypes.sol

Final Commit Hash

6c745a1fb2c5cc77df7fd3106f57db1adc947b75

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	3	0

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: FeeManager#chargeManagementFee fails to init timestamp when treasury == address(0)

Source: https://github.com/sherlock-audit/2025-04-seamless-protocol-leverage-tokens/issues/17

Summary

`FeeManager#chargeManagementFee` is called during token creation to init the fee accrual timestamp to the the current timestamp. Due to the early return when `treasury == address(0)` this does not happen and timestamp is left at zero. The results in huge fees being calculated (from 1970 to current) and applied when the treasury is set.

Vulnerability Detail

LeverageManager.sol#L183-L216

```
        function createNewLeverageToken(LeverageTokenConfig calldata tokenConfig,
        ↪  string memory name, string memory symbol)
            external
            nonReentrant
            returns (ILeverageToken token)
        {

            ... snip

            _setLeverageTokenActionFee(token, ExternalAction.Mint,
            ↪  tokenConfig.mintTokenFee);
            _setLeverageTokenActionFee(token, ExternalAction.Redeem,
            ↪  tokenConfig.redeemTokenFee);
@>          chargeManagementFee(token);

            ... snip

            return token;
        }
```

When creating a token `chargeManagementFee` is called on the empty token to init the timestamp.

FeeManager.sol#L127-L143

```
        function chargeManagementFee(ILeverageToken token) public {
            address treasury = getTreasury();
```

```
            // If the treasury is not set, do nothing. Management fee will continue
             ↪  to accrue
            // but cannot be minted until the treasury is set
@>          if (treasury == address(0)) {
                return;
            }


            // Shares fee must be obtained before the last management fee accrual
             ↪  timestamp is updated
            uint256 sharesFee = _getAccruedManagementFee(token);
            _getFeeManagerStorage().lastManagementFeeAccrualTimestamp[token] =
             ↪  uint120(block.timestamp);


            // slither-disable-next-line reentrancy-events
            token.mint(treasury, sharesFee);
            emit ManagementFeeCharged(token, sharesFee);
        }
```

However when the `treasury == address(0)` the timestamp is never updated. This fails to init timestamp which will reamain at 0.

Impact

Tokens created while `treasury == address(0)` will be charged massive amounts of fees

Code Snippet

FeeManager.sol#L127-L143

Tool Used

Manual Review

Recommendation

Fee management should be reworked with this and the other fee issue in mind. A bare minimum the fee timestamp should init when `chargeManagementFee` is called

Issue M-2: Management fee collection methodology is incorrect when treasury == address(0)

Source: https://github.com/sherlock-audit/2025-04-seamless-protocol-leverage-tokens/issues/18

Summary

When `treasury == address(0)` the methodology in _convertToShares and _getFeeAdjustedTotalSupply causes unfair dilution to users when they mint and redeem.

Vulnerability Detail

LeverageManager.sol#L353-L382

```
    function _convertToShares(ILeverageToken token, uint256
    ↪  equityInCollateralAsset, ExternalAction action)
        internal
        view
        returns (uint256 shares)
    {
        ILendingAdapter lendingAdapter = getLeverageTokenLendingAdapter(token);

@>      uint256 totalSupply = _getFeeAdjustedTotalSupply(token);
        uint256 totalEquityInCollateralAsset =
        ↪  lendingAdapter.getEquityInCollateralAsset();

        ... snip

        Math.Rounding rounding = action == ExternalAction.Mint ?
        ↪  Math.Rounding.Floor : Math.Rounding.Ceil;
@>      return Math.mulDiv(equityInCollateralAsset, totalSupply,
↪  totalEquityInCollateralAsset, rounding);
    }
```

When calculating the number of shares on mint/redeem, the contract uses `_getFeeAdjustedTotalSupply` as the total supply.

FeeManager.sol#L206-L216

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

```
        uint256 duration = block.timestamp - lastManagementFeeAccrualTimestamp;

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

Notice above that the sharesFees are calculated dynamically based on the amount of time that has passed. Since the timestamp is not updated when treasury == address(0), this fee is retroactively applied after minting shares and then destroyed upon redeeming. Take the following example:

Assume there is no treasury address and a 1% fee has accumulated. A user mints 100 shares (1e20). Since they have now minted 100 shares, 1 share worth of management fees are retroactively applied. Due to the shares based distribution, their deposit is immediately diluted.

When shares are redeemed under these conditions, the management fees disappear but during the redemption they are considered in the accounting. Take the same example as above. When those 100 shares are redeemed they would be treated as if there was 101 shares (100 real shares + 1 management share). The user would receive 100/101 of their initial assets and the management share would also disappear. This would effectively cause the management share to be ”burned” and it's assets redistributed to the other shares via inflation.

Impact

New depositors are unfairly diluted, receiving far fewer shares for their collateral. Redeemers lose value when their burns include phantom shares that reduce their actual withdrawal.

Code Snippet

FeeManager.sol#L127-L143

Tool Used

Manual Review

Recommendation

Fee distribution should happen like normal even when there is no treasury set but the fee count should be cached rather than minted.

Issue M-3: Unused shares not returned to user in _redeemAndRepayMorphoFlashLoan and may accumulate to a significant amount.

Source: https://github.com/sherlock-audit/2025-04-seamless-protocol-leverage-tokens/issues/19

Summary

In the `_redeemAndRepayMorphoFlashLoan` function, the `params.maxShares` are transferred from the user to the contract. However, the `leverageManager.redeem` function may use fewer shares than `params.maxShares` to fulfill the redemption request. The difference between `params.maxShares` and the actual shares used is not returned to the user, leading to unused shares accumulating on the contract.

Vulnerability Detail

The issue lies in the `_redeemAndRepayMorphoFlashLoan` function in the `LeverageRouter` contract. The function transfers the maximum number of shares (`params.maxShares`) from the user to the contract:

```
SafeERC20.safeTransferFrom(params.token, params.sender, address(this),
↪  params.maxShares);
```

The `leverageManager.redeem` function is called to redeem the required equity:

```
uint256 collateralWithdrawn = leverageManager.redeem(
    params.token,
    params.equityInCollateralAsset,
    params.maxShares
).collateral;
```

The `leverageManager.redeem` function may use fewer shares than `params.maxShares` to fulfill the redemption request.

Similar issue may happen in the `repay` function of the `MorphoLendingAdapter`, the user transfers an `amount` of the debt asset to the contract. However, if the amount exceeds `maxAssetsToRepay` (the total debt owed), the excess debt asset is not returned to the user. This issue can lead to the accumulation of unused debt assets on the contract, but it may only occur when the function is called directly by the user, as opposed to being invoked by the `LeverageManager`.

Impact

Over time, multiple such events can lead to a significant amount of unused shares being held by the contract.

Code Snippet

https://github.com/sherlock-audit/2025-04-seamless-protocol-leverage-tokens/blob/f394e5aeb7abe66ae628d3afec5df979f7641fb4/leverage-tokens/src/periphery/LeverageRouter.sol#L226-L238

Tool Used

Manual Review

Recommendation

After the `leverageManager.redeem` call, calculate the difference between `params.maxShares` and the actual shares used. Return the unused shares to the user:

```
uint256 unusedShares = params.maxShares - actionData.sharesUsed;
if (unusedShares > 0) {
    SafeERC20.safeTransfer(params.token, params.sender, unusedShares);
}
```

Alternatively, rescue functions can be implemented on contracts where dust amounts may accumulate or funds sent mistakenly.

```
function rescueERC20(address token, address to, uint256 amount) external
↪  onlyPrivilegedRole {
    require(to != address(0), "Invalid recipient address");
    SafeERC20.safeTransfer(IERC20(token), to, amount);
}
```

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.