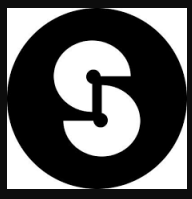
certora

Security Assessment & Formal Verification Report



Seamless ILM

April 5, 2024

Prepared for
Seamless Protocol

Project Summary

Project Scope

Repo Name	Repository	Commits	Compiler version	Platform
ilm	https://github.com/seamless-protocol/ilm	697b389	Solidity 0.8.21	EVM
Ilm (fixes #1)	https://github.com/seamless-protocol/ilm	f480d90	Solidity 0.8.21	EVM
Ilm (fixes #2)	https://github.com/seamless-protocol/ilm	293f560	Solidity 0.8.21	EVM

Project Overview

This document describes the specification and verification of the **Integrated Liquidity Market (ILM)** using the Certora Prover and manual code review findings. The main work was undertaken from **28 January 2024** to **29 February 2024,** with an additional 3 days **(March 25-28)** for fix review.

In the table above, the original code submitted for review is commit hash 697b389. The first round of code fixes is commit hash f480d90 from February 21. The second and final round of fixes reviewed in this engagement is commit hash f480d90 from March 20. The file structure remained the same throughout the three versions of the code base. The following Solidity files are included in our scope:

`src/LoopStrategy.sol`
`src/swap/adapter/WrappedTokenAdapter.sol`
`src/swap/adapter/AerodromeAdapter.sol`
`src/swap/adapter/SwapAdapterBase.sol`
`src/swap/Swapper.sol`
`src/libraries/LoanLogic.sol`
`src/libraries/math/RebalanceMath.sol`
`src/libraries/math/ConversionMath.sol`

`src/libraries/math/USDWadRayMath.sol`
`src/libraries/RebalanceLogic.sol`
`src/storage/AerodromeAdapterStorage.sol`
`src/storage/LoopStrategyStorage.sol`
`src/storage/SwapAdapterBaseStorage.sol`
`src/storage/SwapperStorage.sol`
`src/storage/WrappedTokenAdapterStorage.sol`
`src/tokens/WrappedCbETH.sol`
`src/types/DataTypes.sol`

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora Prover discovered bugs in the Solidity contracts code, as listed below.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	2	2	2
High	0	0	0
Medium	5	5	2
Low	8	8	2
Informational	11	10	6
Total	**26**	**24**	**12**

Detailed Findings

Critical Severity Concerns

C-1. LoanLogic returns the USD value of total collateral in all tokens, not just the collateral token

Impact: High
Probability: High
Category: Logic, AAVE integration, Economical

Description: whenever the strategy needs to perform any operation (rebalance, withdraw, deposit,...) it invokes the method getLoanState from the LoanLogic library to get the current value (in USD) of the total collateral and total debt of the strategy in the Seamless pool. Examining the code, we note that strategy always implicitly assumes that the value of the total collateral reported by the pool is the value of the underlying collateral asset. *However, this is not necessarily true*. Note that:

- AAVE V3 allows *any* user to supply collateral on behalf of another user. In particular, an attacker can "gift" collateral to the lLM loop strategy.
- The method getLoanState is a wrapper around AAVE's `calculateUserAccountData` method which actually returns the sum of the value of all collaterals (in USD) locked in the pool.

Exploit Scenario 1 (gift attack): Assume that we have three distinct ERC20 tokens on Base, whose addresses we shall denote as A,B,C. We further assume that asset A is the collateral asset of the Loop Strategy, and asset B is the debt asset. We shall call asset C the gift asset. Each token is worth 1$. Suppose that:

1. The strategy starts from having 1000 token A deposited, and for simplicity no debt (the situation with rebalancing is similar, just more numerically complicated).
2. Thus, at its original state it has total collateral 1000$, and 1000 total shares.
3. Alice (the attacker) deposits 10000 token A into the Strategy.

4. Since total collateral = 11000$, Alice gets minted 9990 shares and total shares=10990.
5. Alice now *gifts* 10000 token C to the ILM. Note that at this point the value of the total collateral = 21000$.
6. Now Bob (the victim) deposits 10000 token A in the ILM as well. Total collateral is now 31000$, so Bob is minted 5233 shares, with total shares = 16223. But we only have 21000 token A deposited as collateral.
7. Alice now withdraws 9990 shares, which are worth 19089$, leaving the strategy with only 1911 token A locked as collateral.
8. Thus, Bob can withdraw *at most* 1005 of its shares, the rest are permanently locked with the strategy.

Exploit Scenario 2 (volatility bomb): this is another exploit of the previous bug, but this time Alice (the attacker) is targeting the strategy itself using the rebalance mechanism.

In this scenario, Alice chooses token C (the gift token) to be an *extremely* volatile asset. Since the strategy *mistakenly* assumes that it has a large amount of collateral in token A, it tries to increase its leverage to match the target collateral ratio, effectively becoming over-exposed. When the strategy inevitably falls below the liquidation threshold, Alice liquidates it, regaining the "gifted" tokens as well as the collateral belonging to all the other users who deposited funds into the strategy.

Seamless Response: Thanks for reporting this issue. It should be resolved in the latest commit.

Fix Review: The issue is present in the original code and in fix #1. The issue has been successfully resolved in fix #2 by adding the internal method `_getCollateralUSD` to `getLoanState`.

C-2. The logic in UserShareDebt == StrategyDebt is wrong

Impact: High
Probability: High
Category: Logic, AAVE integration

Description: The logic of the code branch

```
Unset
    if (state.debtUSD == shareDebtUSD) {
      // pay back the debt corresponding to the shares
      rebalanceDownToDebt($, state, 0);

      state = LoanLogic.getLoanState($.lendingPool);
      shareEquityUSD = state.collateralUSD - state.debtUSD;
    }
```

is flawed. The strategy wrongfully assumes that if UserShareDebt == StrategyDebt then shareEquityUSD = state.collateralUSD - state.debtUSD. This assumption is incorrect and allows an attacker to drain the protocol and win other users' funds.

There are multiple different paths to enter this code branch with shares and debt values which do not follow the naive assumption. We will outline several of them here.

Exploit Scenario 1: an attacker has the option to annul the entire strategy debt *without interacting with the ILM* by using AAVE's Pool.repay() and then calls redeem() with any amount of shares. The following functions are called rebalanceBeforeWithdraw(), updateState(), and getLoanState(), which return a `state` with `debtUSD == 0`. Now `shareDebtUSD == 0` because the overall debt is zero. At this end `rebalanceBeforeWithdraw()` perform the following check.

```
Unset
if (state.debtUSD == shareDebtUSD) {
            …
          shareEquityUSD = state.collateralUSD - state.debtUSD;
```

and the caller of `redeem()` receives the amount of assets corresponding to `shareEquityUSD`.

The Certora Prover generated a counter-example that exemplifies such a scenario. The prover checked the following property: A user cannot redeem more than deposited, assuming the oracle price.... The trace shows a call to `redeem()` with zero shares when `debtUSD == 0` we see that `shareEquityUSD == 6` and the amount of assets received is `3`.

Exploit Scenario 2: In the method rebalanceBeforeWithdraw() of RebalanceLogic.sol, we have the following case:

```
Unset
    LoanState memory state = updateState($);

    // calculate amount of debt and equity corresponding to shares in USD value
    (uint256 shareDebtUSD, uint256 shareEquityUSD) =
      LoanLogic.shareDebtAndEquity(state, shares, totalShares);

    // if all shares are being withdrawn, then their debt is the strategy debt
    // so in that case the redeemer incurs the full cost of paying back the debt
    // and is left with the remaining equity
    if (state.debtUSD == shareDebtUSD) {
      // pay back the debt corresponding to the shares
      rebalanceDownToDebt($, state, 0);

      state = LoanLogic.getLoanState($.lendingPool);
      shareEquityUSD = state.collateralUSD - state.debtUSD;
    }
```

The problem is that shareDebtAndEquity rounds the number of shares **up**:

```
Unset
    shareDebtUSD = Math.mulDiv(state.debtUSD, shares, totalShares, Math.Rounding.Ceil);
```

so it is possible for shareDebtUSD == state.debtUSD even though shares < totalShares. For instance, suppose the strategy has CollateralUSD = 1,000,095$ and DebtUSD = 95$. Alice (the attacker) owns 99 shares out of a total of 100 shares and Bob owns the remaining share. So equity per share is 10,000$ and the attacker originally had equity of 990,000$. Computing, we

see that shareDebtUSD = (99*95)/100 = 94.05$ which then gets rounded up to 95$=DebtUSD. Thus, after repaying the debt, Alice is left with shareEquityUSD = 1,000,000$ which means that she took 10,000$ of Bob's money.

Exploit Scenario 3: Returning to the method rebalanceBeforeWithdraw(), note that the method updateState could *exit* with debtUSD == 0 while collateralUSD >0. Again, this can happen along multiple code paths:

- First, this might occur if the method rebalanceNeeded returns false. Looking at the definitions of collateralRatioUSD and isCollateralRatioOutOfBounds we see this is possible if collateraRatioTargets.maxForRebalance was set to max_uint256, that is, if the loop strategy is currently configured to avoid debt. In such a case (even if the rounding direction was correct, see H-1) it is possible for a user to redeem the full value of the collateral with less than the total amount of shares since any number multiplied by zero is zero:

```
Unset
    shareDebtUSD = Math.mulDiv(state.debtUSD, shares, totalShares, Math.Rounding.Ceil);
//@audit: if state.debtUSD == 0 then shareDebtUSD == 0 even for shares<totalShares!
```

- Second, it is possible that rebalanceTo would turn a non-zero debt into zero debt in an effort to rebalance down and decrease exposure (see concrete example found by the Prover below).

Regarding the first subcase, the Certora Prover found 3 scenarios that demonstrate possible attacks. Both examples are detailed numerical examples leading to a state where `debtUSD == 0`, thus causing the redeemer to fetch all the collateral.

Numerical example 1

Initial state
price = 4, decimals = 22, totalCollateralBase = 1.5 * 10^24
collateralUSD = (1.5 * 10^24 * 4) / 10 ^ 22 = 600
totalDebtBase = 3
debtUSD = 0 => collateralRatioUSD = max_uint256 => rebalanceNeeded() = false

equityUSD = 600 - 0 = 600
targets = (1, 1, max_uint256, 52, 0)
shares = 50, totalSupply = 150

redeem()
updateState() does nothing
shareEquityUSD = 200, shareDebtUSD = 0
state.debtUSD == shareDebtUSD => the redeemer fetches all the collateral

Post state
totalCollateralBase = 0, totalDebtBase = 3

Numerical example 2 shows that `debtUSD` can be zero even if `debtBase > 0`. The run trace shows the following.

```
debtUSD = (debtBase * price) / 10 ^ decimals = (9 * 12) / 10^12 = 0.
```

Numerical example 3

Here a user calls redeem() with **zero** shares gets **all** the collateral.

Init state
maxForRebalance = 2500000.1625 * 10^8 (not equal to max_uint256)
decimals = 13, price = 1
totalCollateral = 15.999999 * 10^13 = 15 USD (round down)
totalDebt = 1.0000000000002 * 10^13 = 1 USD (round down)
currentLiquidationThreshold = 4002
maxWithdrawAmount = 13
rebalanceDown() call repay(0.9999999999994 * 10^13) where margin = 5000000

Post rebalanceDown()
totalCollateral = 14.999 * 10^13 = 14 USD
totalDebt = 8 = 0 USD

Now we enter the code of the above-mentioned branch.

Recommendation: change the computation of `shareEquityUSD` so that it will return the right amount.

Seamless Response: We acknowledge the issue. The attack scenarios are caused by this line:

```
Unset
shareEquityUSD = state.collateralUSD - state.debtUSD.
```

This has been resolved in the latest commit.

Fix Review: The issue is present in the original code and in fix #1. It has been successfully resolved in `fix #2` by removing the problematic code branch.

Medium Severity Concerns

M-1. Not checking for stale prices

Impact: Medium
Probability: Medium
Category: Chainlink integration

Description: The ILM code interacts with the price feed via AAVE's Oracle interface which is missing a check if the return value indicates stale data. This could lead to stale prices according to the Chainlink documentation, see [1] and [2]. Note that in known (extreme) historical cases, chainlink price feed has seen delays as large as six hours.

Impact: the strategy might be relying on outdated information which can lead e.g., to over-exposure to dangerous assets.

Seamless Response: We acknowledge the issue and plan to fix it**.** We will advise the community of this and look to make a fix to the oracle logic (AaveOracle.sol) used by the lending pools.

M-2. Missing min/max boundary check

Impact: High
Probability: Very Low
Category: Chainlink integration

Description: Chainlink price feeds have in-built minimum & maximum prices they will return. Thus if during a flash crash (e.g., the LUNA incident), bridge compromise, or depegging event, a certain asset's value falls below the price feed's minimum price, the price feed will continue to report the (now incorrect) minimum price. As a result, the ILM strategy could find itself over-exposed to a risky asset.

Recommendation: The events we described are (thankfully) rare, but can have very bad consequences to protocols involved which do not have proper safeguards in place.

To help mitigate such an attack on-chain, smart contracts could check that minAnswer < receivedAnswer < maxAnswer. Alternatively, this attack could also be mitigated via off-chain monitoring, which compares Chainlink's reported price with other off-chain sources (e.g., centralized exchanges and/or liquid indexes).

Seamless Response: We acknowledge the issue and plan to fix it**.** We will advise the community of this and look to make a fix to the oracle logic (AaveOracle.sol) used by the lending pools.

M-3. rebalanceNeeded() returns true although rebalance is not required

Impact: Medium
Probability: High
Category: Logic

Description: rebalanceNeeded() returns `true` when both `collateralUSD` and `debtUSD` are zero although rebalance is not needed when there is no collateral. It stems from collateralRatioUSD() that returns `max_int` when `debtUSD == 0` without checking `collateralUSD`.

```
Unset
function collateralRatioUSD(uint256 collateralUSD, uint256 debtUSD)
{
        ratio = debtUSD != 0 ? collateralUSD.usdDiv(debtUSD) : type(uint256).max;
}


function rebalanceNeeded(uint256 collateralRatio, CollateralRatio memory
collateraRatioTargets) internal pure returns (bool) {
        return (
            collateralRatio < collateraRatioTargets.minForRebalance
                || collateralRatio > collateraRatioTargets.maxForRebalance
        );
```

Certora Prover ran Property rebalanceNeeded() should return false when called immediately after rebalance() and found a counter-example demonstrating such a scenario.

Exploit Scenario: An attacker calls `rebalance()` multiple times, and the protocol loses funds.

Seamless Response: Thanks for reporting this issue. It should be resolved in the latest commit.

M-4. getLoanState() does not check totalDebtUSD

Impact: Medium

Probability: Low

Category: Logic

Description: getLoanState() assigns zero to `debtUSD` when `totalCollateralUSD == 0` without checking `totalDebtUSD`.

```
Unset
    if (totalCollateralUSD == 0) {
      return LoanState({
        collateralUSD: 0,
        debtUSD: 0,
        maxWithdrawAmount: 0
      });
    }
```

Exploit Scenario: Immediately after **bad-debt liquidation**, the collateral goes to zero but the debt is still positive.

Seamless Response: We acknowledge the issue and have changed `debtUSD` default value.

Fix Review: The issue is present in the original code and in fix #1. It has been successfully resolved in fix #2 by changing the code fragment above to

```
Unset
    if (totalCollateralUSD == 0) {
      return LoanState({
        collateralUSD: 0,
        debtUSD: 0,
        maxWithdrawAmount: 0
      });
    }
```

M-5. Centralization Risk

Impact: High
Probability: Low
Category: Centralization

Description: There are multiple privileged rules in the code, access to which would allow a hacked or malicious party to jeopardize critical functions in the projectthe code of the ILM contains a few privileged rules:

- `UPGRADER_ROLE (LoopStrategy.sol)`
- `PAUSER_ROLE (LoopStrategy.sol)`
- `MANAGER_ROLE (LoopStrategy.sol)`
- `DEFAULT_ADMIN_ROLE (LoopStrategy.sol)`
- `UPGRADER_ROLE (Swapper.sol)`
- `STRATEGY_ROLE (Swapper.sol)`
- `MANAGER_ROLE (Swapper.sol)`
- `DEFAULT_ADMIN_ROLE (Swapper.sol)`
- `DEPOSTER_ROLE (WrappedERC20PermissionedDeposit.sol)`
- `DEFAULT_ADMIN_ROLE (WrappedERC20PermissionedDeposit.sol)`

These rules have access to privileged methods in the contract that can lead to a loss of funds or disrupt critical functionality. Thus, they represent a single point of failure and should be carefully protected.

Recommendation: Firstly, when possible, privileged functions that change critical parameters should emit events and have timelocks. Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with time locks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. Secondly, we suggest allowing only multi-signature wallets to call the function to reduce the likelihood of an attack. Thirdly, we recommend taking extra precautions with the various `DEFAULT_ADMIN_ROLE` and use OZ's AccessControlDefaultAdminRules library to enforce additional security measures regarding this role.

Seamless Response: Sensitive roles will be assigned to the Seamless governance timelock. Less sensitive roles such as the PAUSER_ROLE may be assigned to governance-approved multi sigs.

Low Severity Concerns

L-1. Missing check for L2-sequencer state

Impact: Medium
Probability: Low
Category: Optimistic Rollups

Description: The ILM is deployed on the Base L2, which means that occasionally it is possible for the sequencer to go down. This has happened in the past and may happen in the future. In such a situation the usual flow of DeFi activity on Base is somewhat “warped” (i.e., there aren’t many arbitrageurs who can continue to operate and keep, Chainlink’s price feeds may not update even though a large deviation in price has occurred, bad loans that need to be liquidated persist, etc). Thus, rebalancing or borrowing under these circumstances can have unforeseen consequences.

Recommendation: prepare for this eventuality - add a check for L2 sequencer state and consider how the strategy should behave under such circumstances.

Seamless Response: Acknowledged, we should implement additional sequencer health checks in addition to the PriceOracleSentinel.

L-2. Replace the overly-complicated logic of the view function

Impact: Low
Probability: Medium
Category: Logic

Description: the view functions estimateSupply and estimateWithdraw (invoked via the wrapping methods previewDeposit and previewRedeem) are meant to provide user with information (accurate as possible) regarding the result of their interactions (i.e., deposit/redeem) with the Loop Strategy contract. However, the code they contain is essentially a rather inaccurate approximation to what the actual deposit/redeem methods do because it cannot take the conditions in the DEX route into account. It is also hard to maintain and update and prone to error since it has to be updated with every change to the core logic.

Recommendation: replace these functions with a wrapper which calls the actual deposit/redeem method and then reverts the result. While not formally being a "view" function, it is effectively the same and can be called off-chain via an RPC call.

Seamless Response: We acknowledge the issue, but we want to note that currently off-chain static call directly to the deposit/redeem functions would effectively have the same result as your proposal. We would like to keep previewDeposit and previewRedeem functions for the potential on-chain use and compliance with ERC4626.

L-3. No sanity check in strategy initialization

Impact: Medium
Probability: Low
Category: Missing Input Validation

Unlike setCollateralRatioTargets(...) and setRatioMargin(), the initialization method of the ILM LoopStrategy contains no sanity checks for `collateralRatioTargets` and for `ratioMargin`. Similarly, there is no check that the oracle address supplied actually matches the one used by the Seamless pool, even though it can easily be derived via a getter method and does not to be passed as a variable during initialization at all. The following Certora Prover report shows counter-examples of CVL invariants that exemplify the former issues.

Seamless Response: We acknowledge the issue and added validation checks in the initializers.

Fix Review: The issue is present in the original code and in fix #1. It has partially resolved in fix #2 by adding the internal methods _validateCollateralRatioTargets(...) and _validateRatioMargin(...). However, the oracle address is still provided as a parameter instead of being derived from the poolAddressProvider (as is done, e.g., in LoanLogic).

L-4. An attacker can sandwich rebalancing

Impact: Medium
Probability: Low
Category: Blockchain, Economical, Design

Description: MEV is a basic problem in the trading world with varying impact on different DeFi primitives. In some cases, it is arguably not a problem at all while in others it can be the crux of the problem the system is aiming to solve (e.g., the raison d'etre of any TWAMM system is essentially to mitigate it). Since Seamless's Integrated Liquidity Market (ILM) is a type of automatic investment strategy (as opposed to a DEX or a lending/borrowing platform), we argue that the MEV question here is of paramount importance.

- While researching the system and its interface, we learned from the ILM team that the strategy was initially designed with Optimism's *legacy* model in mind which created a block for each transaction, processing new transactions in a first-come-first-serve manner. Thus, Optimism legacy chain had no mempool and therefore a greatly reduced MEV attack surface (though it doesn't diminish them entirely: back-running is still an issue as demonstrated in a recent preprint by Bagourd & Francois).
- However, the new Optimism chain (i.e., post-Bedrock) introduced a key difference. In this new version, Optimism creates blocks on a fixed interval - and therefore needs a mempool to store pending transactions until they are included in a block (we do remark however that the mempool is kept private in an effort to reduce the MEV attack surface).
- The Base chain (on which Seamless operates) is based on this new OP-stack which replaced the FCFS Sequencer with a fee market with base/priority fees (EIP-1559), opening the door to potential exploits. There isn't currently a published study on the MEV potential of post-Bedrock Optimism.
- Furthermore, in both cases (legacy/Bedrock) it is possible for users to bypass the Sequencer and submit transactions (referred to as 'deposit' in Optimism terminology) directly to the Canonical Transaction Blockchain (CTC). Such L1 -> L2 transactions are submitted via the usual mainnet mempool.
- Since MEV is not technically illegal and extrapolating from the evidence in mainnet and other optimistic L2 Rollups, we believe that it is likely that sophisticated players (e.g.,

Hedge funds,...) would take advantage of the ILM strategy when the technical possibility to do so exists.

- The basic way the Base chain protects users from MEV is by deploying a private mempool (called the "transaction mempool"), i..e, transactions are not shared with anybody until they are actually included in a block.
- This solution has two issues: first, it obviously requires us to trust the Sequencer. Second, when the ILM strategy is unbalanced this does not provide protection. For example, any EOA (even one which has no shares in the strategy) can trigger rebalance() which executes a trade via a sequence of predetermined routes with known slippage thresholds.
- Thus, an attacker can know in advance of the existence of a transaction even though the transaction pool is private.

Seamless Response: We acknowledge the possibility of sandwiching DEX swaps during rebalances, with MEV capture up to the configured slippage threshold. We expect this MEV to be relatively low and we intend to implement mechanisms to limit it even further in the future. One example of a solution is to use a dutch auction to have MEV bots compete against each other to drive down the MEV cost incurred by the strategy.

L-5. If collateralAmountAsset == 0 we should cancel the rebalancing process

Impact: Low

Probability: Low

Category: Logic

Description: Suppose that the collateral token is A and the debt token is B. In L.#441-#466 of rebalanceUp, the strategy borrows a `borrowAmountAsset` of token B from the pool and attempts to swap it via the swapper's DEX route into a certain amount ("`collateralAmountAsset`") of A-tokens. In case this quantity turns out to be zero, we break and exit the loop. Note however that in this case we are left with B-tokens (and a corresponding debt) which we never use, thus leading to a loss of equity.

Unset

```
if (borrowAmountAsset == 0) {
  break;
}

// borrow _assets from lending _pool
LoanLogic.borrow($.lendingPool, $.assets.debt, borrowAmountAsset);

// approve _swapper contract to swap asset
$.assets.debt.approve(address($.swapper), borrowAmountAsset);

// exchange debtAmountAsset of debt tokens for collateral tokens
uint256 collateralAmountAsset = $.swapper.swap(
  $.assets.debt,
  $.assets.collateral,
  borrowAmountAsset,
  payable(address(this))
);

if (collateralAmountAsset == 0) {
  break;
}
```

```
// collateralize _assets in lending _pool
_state = LoanLogic.supply(
  $.lendingPool, $.assets.collateral, collateralAmountAsset);
```

Similarly in reBalanceDown L.#525-#534 we attempt to withdraw some A-tokens and exchange them for B-tokens in order to repay a portion of our debt:

```
Unset
if (collateralAmountAsset == 0) {
  break;
}

uint256 borrowAmountAsset =
  withdrawAndSwapCollateral($, collateralAmountAsset);

if (borrowAmountAsset == 0) {
  break;
}

// repay debt to lending _pool
state = LoanLogic.repay($.lendingPool, $.assets.debt, borrowAmountAsset);
```

However, if the amount of B-tokens we manage to receive is zero there is no reason for the strategy to keep an untapped supply of A-tokens instead of providing it as collateral for our loan. Since in that case we have decreased the health factor of the Strategy's loan without any gain to match. The situation in rebalanceDownToDebt (L.#582-#595) is identical.

Recommendation: in all cases one can perform the borrowing, swapping and supply/repay in a new call frame, and revert the changes and exit if `collateralAmountAsset != 0` but `borrowAmountAsset == 0`

Seamless Response: We acknowledge the issue, and we will prepare a fix in the future.

L-6. Decimals() should match the Seamless pool configuration

Impact: High
Probability: Low
Category: Logic, AAVE Integration

Description: In many places and across many files (LoopStrategy.sol, LoanLogic.sol, RebalanceLogic.sol, Swapper.sol) the ILM code queries the number of decimals of an asset using the decimals() ERC20 method. However, it is very possible that this number would not match the decimals used by the AAVE pool which hard-codes the configuration of each asset. In such a case, the value computation would be completely wrong.

Implication: In the worst-case scenario, this can cause the strategy to become liquidated.

Recommendation: Fix these parts in the code to work with AAVE pool parameters for each asset.
Seamless Response: Thank you for reporting this, and we will prepare a fix in the future.

L-7. Unnecessary rebalancing can occur and even change the collateral ratio in an undesired direction

Impact: Low
Probability: Low
Category: Logic, Gas

Description: The function rebalanceTo() doesn't consider the margin when deciding about rebalancing:

```
Unset
    if (ratio > targetCR) {
      return rebalanceUp($, state, ratio, targetCR);
    } else {
      return rebalanceDown($, state, ratio, targetCR);
    }
```

Note that the methods rebalanceUp() and rebalanceDown() must execute *at least* one rebalancing round because they use a do-while loop. This is bad for the strategy in two ways: first, it wastes gas and second, it can lose a bit of equity (by virtue of paying DEX fees) - all in a situation where no rebalancing was actually required to take place. The Prover call trace below demonstrates such an instance.

Numerical example
Details:

Max slippage = 1%

collat = 100010000

debt = 100000000

CR = 100010000

target = 200000000,

minForRebalance = 100010001,

maxForRebalance = 200000000

rebalanceDown():

withdraw(9992)

swap(9999) = 9992

repay(9992)

collat = 100000001

debt = 099990008

CR = 100009994

Recommendation: Consider fixing this so that unnecessary rebalance won't happen.

Seamless Response: Thank you for reporting this, and we will prepare a fix in the future.

L-8. No borrow cap leads to incorrect calculation

Impact: Low
Probability: Medium
Category: AAVE integration

Description: In AAVE, the situation in which there is no borrowCap on a certain asset is designated by setting the borrow cap to zero. However, the ILM code of the method getAvailableAssetSupply is not aware of this and treats this case as if there was nothing to borrow, which leads to avilableUntilBorrowCap and therefore also availableAssetSupply being zero, which leads the strategy to believe that it cannot rebalance up even in situations where it should and leaving the depositors under-exposed to the intended asset.

```
Unset
        uint256 totalBorrow = _getTotalBorrow(reserveData);
        uint256 borrowCap = reserveData.configuration.getBorrowCap();
        uint256 assetUnit = 10 ** reserveData.configuration.getDecimals();
        uint256 avilableUntilBorrowCap = (borrowCap * assetUnit > totalBorrow)
            ? borrowCap * assetUnit - totalBorrow
            : 0;

        uint256 availableLiquidityBase =
            asset.balanceOf(reserveData.aTokenAddress);

        availableAssetSupply =
            Math.min(avilableUntilBorrowCap, availableLiquidityBase);
        return availableAssetSupply;
```

Recommendation: fix the code to account for the case where there is no borrow cap.

Seamless Response: Thanks for reporting this issue. It should be resolved in the latest commit.

Informational Concerns

I-1. Unused storage variable

Description: `LoopStrategyStorage.layout.usdMargin` is unused.

Seamless Response: Acknowledged and fixed.

I-2. Non-uniform naming convention

Description: the methods debt() and collateral() should perhaps be named `debtUSD()` and `collateralUSD()` to fit with the naming convention in `LoopStrategy.sol`.

Seamless Response: Acknowledged and fixed.

I-3. Typo in a comment in L.#17 of RebalanceMath.sol

Description: in the comment in line 17:

```
Unset
// @param debtUSD debt valut in USD
```

Replace `valut` with `value.`

Seamless Response: Acknowledged and fixed.

I-4. Gas optimization - rebalance should exit

Description: rebalance() should exit rather than revert when rebalance is not required.

Seamless Response: Acknowledged, but will not be changed, due to easier tracking of unneeded rebalance keeper calls (keepers will likely simulate transaction and fail before submitting on chain), and negligible gas savings.

I-5. Safety enhancement - always rebalance after setting CR targets

Description: A safety suggestion: setCollateralRatioTargets() should call rebalance() rather than MANAGER_ROLE calling it off-chain.

Seamless Response: Acknowledged and fixed.

I-6. Missing input validation

Description: setCollateralRatioTargets() should enforce that `maxForDepositRebalance >= target`. The current implementation allows maxForDepositRebalance to be zero, and `minForWithdrawRebalance` could be `max_uint256`.

Seamless Response: Acknowledged and fixed.

I-7. Type in a function name: getRatioMagin()

Description: getRatioMagin() ⇒ `getRatioMargin()`

Seamless Response: Acknowledged and fixed.

I-8. If the strategy is sufficiently in debt, all deposits revert

Impact: High
Probability: Low
Category: Logic

Description: All deposit methods in LoopStrategy.sol invoke the internal methods _deposit of which in turn calls the function totalAssets(). But totalAssets() is the same as equity() which invokes equityUSD():

```
Unset
function equityUSD() public view override returns (uint256 amount) {
  LoanState memory state =
    LoanLogic.getLoanState(Storage.layout().lendingPool);
  return state.collateralUSD - state.debtUSD;
}
```

Impact: If the strategy is sufficiently "under-water" so that debtUSD>collateralUSD any attempts by users to deposit funds and save it from liquidation would revert.

Seamless Response: If the strategy is in the position where debt is bigger than collateral, we don't want to allow new deposits. The point of the strategy is that users do not manage collateral ratios directly, keepers should rebalance the strategy long before liquidation occurs.

I-9. `constants` should be defined rather than using magic numbers

Description: In AerodromeAdapter, replace the number 10 with a deadline constant:

```
Unset
block.timestamp + 10 //@audit replace 10 with a constant (e.g., deadline)
```

Seamless Response: Acknowledged and will be fixed.

I-10. `decimals()` should be of type `uint8`

Impact: Low
Probability: Medium
Category: Gas

Affected code:

- src/LoopStrategy.sol

```
# File: src/LoopStrategy.sol

LoopStrategy.sol:620:         uint256 underlyingDecimals =
```

- src/libraries/RebalanceLogic.sol

```
# File: src/libraries/RebalanceLogic.sol

RebalanceLogic.sol:227:         uint256 underlyingDecimals =
```

- src/libraries/math/ConversionMath.sol

```
# File: src/libraries/math/ConversionMath.sol
```

```
ConversionMath.sol:20:         uint256 assetDecimals

ConversionMath.sol:32:         uint256 assetDecimals
```

- src/libraries/math/RebalanceMath.sol

```
# File: src/libraries/math/RebalanceMath.sol

RebalanceMath.sol:88:         uint256 collateralDecimals
```

Seamless Response: Thank you for reporting this issue.

I-11. The computation getMaxBorrowUSD is wrong

Impact: Informational
Probability: Medium
Category: Best Practice

Description: When computing the method getMaxBorrowUSD, we invoke the AAVE method getUserAccountData which in turn calls calculateAvailableBorrows with totalCollateralBase as a parameter. However, totalCollateralBase is the total value of all collaterals - not just the value of the collateral in the collateral token (related: see bug C-1 in this document). Thus we are basically over-estimating the borrowing ability of the protocol.

Recommendation: fix the code to the computation of getMaxBorrowUSD to account only for the value of the chosen collateral token.

Seamless Response:

Formal Verification

Assumptions and Simplifications Made During Verification

General Assumptions

A. Any loop is unrolled to a single iteration at most.
B. All contracts and libraries beyond the Project Scope are summarized, i.e., replaced by an abstracted version of the original Solidity code. In particular:
    a. We use a simplified version of Aave Pool functions: borrow(), repay(), supply(), withdraw() and simplified_getUserAccountData().
    b. We use a simplified version of executeSwap()
    c. For some properties we ignore the concrete implementation of WrappedERC20PermissionedDeposit.withdraw() and deposit(),ERC4626Upgradeable._withdraw(), ERC20Upgradeable._mint() and ERC20.approve()
C. We employ under-approximation abstraction - we replace external values with fixed constants. In particular, we restrict asset price, decimals are replaced with constants.
D. Token transfer summarization - When a transfer of any ERC20-based tokens occurred, instead of using a contract implementation, we used ghost mapping to monitor and store the relevant transfers.
E. Code refactoring and explicit summarizations of internal parts of the code - We refactor the code and replace nonlinear mathematical functions with formal-friendly implementations.

Formal Verification Properties

Notations

✅Indicates the rule is formally verified.
❌Indicates the rule was violated when it checked the initial Solidity version.

Properties that uncovered bugs

1. ❌ A user cannot redeem more than deposited, assuming the oracle price is stable. Uncovered C-2

2. ❌ `rebalanceNeeded()` should return false when called immediately after `rebalance()`. Uncovered C-2
3. ❌ Equity per share cannot decrease after `redeem()`. Uncovered M-3
4. ❌ Invariant: `collateralRatioTargets` are valid. Uncovered: L-3
5. ❌ Invariant: `ratioMargin <= 1 USD`. Uncovered: L-3
6. ❌ Invariant: `minForRebalance <= collateralRatio <= maxForRebalance`.Uncovered: I-5
7. ❌ rebalance() changes the collateral ratio in the desired direction - towards the target. Uncovered L-7

More properties

1. ✅Redeem function reduces the share balance of owner by the share amount specified while calling the function

```
Unset
redeem(e, shares, receiver, owner, minUnderlyingAsset);
```

2. ✅Deposit function increases the receiver's balance by the shares amount it returns if the total supply is greater than or equal to the receiver's balance.

```
Unset
shares = deposit(e, assets, receiver, minSharesReceived);
```

3. ✅rebalance() does change the protocol equity.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.