certora

Security Assessment & Formal Verification Report

EulerEarn

July 2025

Prepared for Euler

certora

Table of Contents

certora

Project Summary

Project Scope

Project Name	Repository	Commit Hash	Latest Fix Commit	Platform
EulerEarn	https://github.com/euler-xyz/euler-earn	a73c52b	685ee26	EVM

Overview

This document describes the specification and verification of **EulerEarn** using the Certora Prover. The work was undertaken from **June 26, 2025** to **July 15, 2025**

The following contract list is included in our scope:

- src/libraries/**
- src/EulerEarn.sol
- src/EulerEarnFactory.sol
- src/PublicAllocator.sol

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. During the verification process, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

EulerEarn is an **ERC4626-compliant** yield aggregator. Based on Metamorpho, it is meant to integrate seamlessly with the Euler ecosystem by replacing Morpho Blue markets with a set of accepted ERC-4626 strategies as well as introducing EVC and permit2 compatibility. The target users of EulerEarn are liquidity providers who want to earn from borrowing interest without having to

actively manage the risk of their position. EulerEarn gives these users the ability to deposit a single asset into EulerEarn and receive vault shares representing their ownership in an actively managed portfolio of multiple ERC4626 vaults (referred to as "strategies" from hereafter) and users can supply or withdraw assets at any time, depending on the available liquidity in the enabled strategies. The active management of the deposited assets is the responsibility of a set of four different roles:

- **Owner**: Ultimate protocol control - sets all roles, manages fees/recipients, controls timelock parameters, and has override access to all other role functions.
- **Curator**: Vault management authority - approves new markets via supply caps, initiates market removals, and can revoke pending changes (also inherits Allocator permissions).
- **Allocator**: Active fund management - controls supply/withdraw queue ordering and executes reallocation strategies across approved vaults.
- **Guardian**: Emergency response role - can revoke pending timelock changes, guardian appointments, market caps, and removal processes to prevent harmful updates.

There are also emergency procedures in place to deal with broken or compromised strategies and to remove a malicious allocator or curator.

Change made from Metamorpho

The EulerEarn repo is a fork of metamorpho v1.1 and public allocator, inspired by silo vaults, with the following changes:

- uses a set of accepted ERC4626 vaults as strategies instead of Morpho Blue markets;
- adds EVC and EulerEarn strategy compatibility;
- adds permit2 compatibility;
- implements EVK-style VIRTUAL_AMOUNT conversions;
- implements zero shares and zero assets protection on deposits and redeems;
- implements internal balance tracking that prevents EulerEarn share inflation;
- adds reentrancy protection;
- removes the skim function and related functionality;
- removes ERC-2612 permit functionality;
- removes Multicall functionality.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	1	1	1
Low	5	5	3
Informational	1	1	1
Total	7	7	5

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
M-01	Share Inflation Attack via Public Allocator	Medium	Fixed
L-01	EVC / Multicall Incompatibility	Low	Fixed
L-02	Use `call()` Instead of `transfer()` for ETH Transfers	Low	Fixed
L-03	Potential Deposit Harvest Withdraw attack if new vault designs are added	Low	Not fixed
L-04	Blacklist may completely block removal of a strategy	Low	Fixed
L-05	Inclusion of standard ERC4626 as strategies requires special care	Low	Not fixed

Medium Severity Issues

M-01. Share Inflation Attack via Public Allocator

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: **PublicAllocator.sol#L96**	Status: Fixed	Violated Property: -

Description:

An attacker might be able to exploit the `PublicAllocator.reallocateTo()` function to steal funds through a share inflation attack on a managed vault. The attack manipulates the share price of an under management target vault (another EulerEarn vault) before reallocating assets to it, allowing the attacker to reallocate funds to the inflated vault and eventually steal funds or at least freeze other user funds at no cost.

Preconditions:
- The EulerEarn vault already has some assets under management.
- The EulerEarn curator just added a new vault (an EulerEarn vault) under management with a balance of zero.
- The PublicAllocator's flow caps (maxIn and maxOut) are configured to permit some reallocation.

Even with EulerEarn vaults inflation attack protection, the fact that every part of this POC can happen within a single transaction makes it potentially vulnerable. This allows the attacker to make a flash loan and directly access vault funds through reallocation.

EulerEarn will likely support other vault designs in the future. Depending on the "inflation attack" protection implemented, this bug could become a high-severity issue.

Exploit Scenario:

The POC can be added to your test suite: link

Recommendations:

In practice, this exploit is super hard to reproduce because of the specific preconditions required, and it's also very hard to make it profitable.

One potential fix could be to revert if the Curator tries to add an empty vault, which forces the vault bootstrapping. And also maybe revert when the reallocation results in an empty vault.

Customer's response: Fixed by adding internal balance tracking and reverts on zero shares/assets on deposit/redeem. link
Fix Review: Fix validated ✅

Low Severity Issues

L-01. EVC / Multicall Incompatibility

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: multicall.sol/#L27-L30	Status: Fixed	Violated Property: -

Description: The issue occurs when a user calls `multicall()` through EVC. After the ***Context ERC2771 multicall attack***, OpenZeppelin patched Multicall by appending the context (_msgSender()) to each call data in multicall. This `context` extracted from `msg.data` is meaningless in EVC's context since it's designed for ERC2771.

Impact:

- **Denial of Service**: Legitimate multicall usage will always fail if called through the EVC.
- **Corrupted calldata**: The sender can accidentally create valid calldata that:
  - Forms a valid function selector
  - Has proper parameter encoding
  - Calls a function with unintended parameters; this is highly unlikely but theoretically possible.

Recommendations: Instead of using OpenZeppelin's implementation, use Solady Multicall. Or use the old OpenZeppelin's multicall.

Customer's response: We were planning to acknowledge this issue, considering that the EVC is a multicall in itself, making no point to use the EVC batching method with the original multicall contract. However, after other fixes, our code now exceeds 24kb, and hence we removed the inheritance from the Multicall completely. (commit)
Fix Review: Fix validated ✅

L-02. Use `call()` Instead of `transfer()` for ETH Transfers

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [publicallocator.sol/#L89]	Status: Fixed	Violated Property: -

Description: The `transferFee` function uses the deprecated `transfer()` method to send ETH to fee recipients. This method has a hardcoded gas limit of 2300 gas, which may cause failures when the recipient is a smart contract.

Recommendations: Replace `transfer()` with a more robust `call()` pattern:

```
function transferFee(address vault, address payable feeRecipient) external
onlyAdminOrVaultOwner(vault) {
    uint256 claimed = accruedFee[vault];
    accruedFee[vault] = 0;

    (bool success, ) = feeRecipient.call{value: claimed}("");
    if (!success) revert ErrorsLib.TransferFailed();

    emit EventsLib.TransferAllocationFee(_msgSender(), vault, claimed, feeRecipient);
```

Customer's response: Fixed. (commit)

Fix Review: Fix validated ✅

L-03. Potential Deposit-Harvest-Withdraw attack if new vault designs are added

Severity: **Low**	Impact: **High**	Likelihood: **Very Low**
Files: [EulerEarn.sol#L832-L834]	Status: Not Fixed	Violated Property: -

Description:

EulerEarn is potentially vulnerable to deposit-harvest-withdraw attacks, where an attacker exploits timing gaps in interest accrual to extract value from existing depositors. This attack involves:

- Timing deposits immediately before interest/reward distribution events
- Harvesting rewards that were generated by other depositors' capital over time
- Withdrawing quickly after rewards are distributed, capturing disproportionate value

Current Risk Assessment: EulerEarn currently integrates only with eVaults and other EulerEarn vaults, which are designed to be resistant to this attack vector. However, future vault integrations may introduce this vulnerability if they implement different reward distribution mechanisms.

Recommendations:

A good solution would be to add a mechanism that distributes rewards linearly over time. However, because this solution delays reward distribution, the Euler team stated that they don't want this fix.

We recommend that the Euler team carefully review the **ERC4626.previewRedeem()** implementation of any newly added vault to ensure that it continually returns the most up to date value.

Customer's response: This code was forked and specifically prepared not to implement the recommended mechanism that was present in the previously implemented version of the euler-earn. Hence, the recommended solution is not feasible and will not be implemented.

Fix Review: OK not to fix in the current configuration. The second part of the recommendation still holds.

L-04. Blacklist may completely block removal of a strategy

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files:	Status: Fixed	Violated Property: -

Description: Suppose that we have a compromised or malicious strategy A and we wish to remove it. As explained in the emergency procedure section of the EulerEarn repo:

- The only way to remove a strategy[1] is to call the EulerEarn function submitMarketRemoval, which first requires setting the market cap to zero.
- In order to do so, the Curator must call the function submitCap on the strategy with _newSupplyCap set to zero.

- The execution necessarily invokes the internal function _setCap, which enters the else branch and invokes the revokeApprovalWithPermit2 function on the asset (with the spender address being the address of our strategy A):

```
JavaScript
function revokeApprovalWithPermit2(IERC20 token, address spender,
address permit2) internal {
        if (permit2 == address(0)) {
            SafeERC20.forceApprove(token, spender, 1);
        } else {
            IAllowanceTransfer(permit2).approve(address(token),
spender, 0, 0);
        }
    }
```

[1] The same (or similar) is also true for removing a strategy from the withdrawQueue using updateWithdrawQueue.

- Now, suppose for instance, the underlying asset has put our strategy on its blacklist and reverts the approval call.
- In such a case, neither the Curator nor the Owner has no way to actually remove the strategy here.

Recommendation: add an emergency removal procedure (perhaps invoked by an account with higher privilege, e.g., the owner) that allows the removal of a broken/malicious market without the need for extra assumptions on the behavior of external contracts.

Customer's response: Fixed. (commit)

Fix Review: Fix validated ✅ This edge case was handled by adding a try/catch.

L-05. Inclusion of standard ERC4626 as strategies requires special care

Severity: **Low**	Impact: **High**	Likelihood: **Very low**
Files:	Status: Acknowledged	Violated Property: -

Description: This is a design/composability issue. To explain it, note that the way the standard OZ implementation of ERC4626 deals with a share inflation attack is by making such an attack unprofitable for an attacker, which will discourage anyone from actually inflating the share price.

An attacker would need to invest a certain amount of funds in order to inflate the price, and that amount must be greater than any loss caused due to rounding to any future depositor. However, the math underpinning this defense is based on some key assumptions which are broken in the case of EulerEarn and its strategies since the attacker may manipulate both sides repeatedly causing EulerEarn to deposit or reallocate funds into the vulnerable strategy.

We emphasize that the issue is currently only theoretical and poses no concrete risk - from conversations with the Euler Team it follows that the only types of strategies currently allowed[2] in EulerEarn are EVK vaults (which have custom virtual amounts defense) and ERC4626 vaults which have been specifically vetted to ensure that they cannot be drained by an attacker via borrow and/or flashloan.

Recommendation: We recommend documenting this issue clearly and carefully testing any third-party ERC4626 vault for integration with EulerEarn. If possible, we further suggest adding a max slippage parameter or constant and a slippage check whenever EulerEarn deposits or reallocates funds into an ERC4626 strategy.

Customer's response: We acknowledge this issue and only plan to improve the documentation: https://github.com/euler-xyz/euler-earn/commit/e06c2f2095d639cfb1269acbfdf35aa6a9a3c469

[2] The types of strategies are enforced by the perspective in the EulerEarnFactory.

As it is today, EulerEarn is meant to be used with other EulerEarn vaults and the EVK vaults as strategies. In order to achieve this, EulerEarn vaults have a special `isStrategyAllowed` allowed check implemented that is used both in the `submitCap` and the `acceptCap` functions. This check is meant specifically not to allow integration with any standard (third-party) ERC4626 implementation. We don't want EulerEarn to integrate with any ERC4626 because this may be unsafe to do so.

We are aware that integrating with a third-party ERC4626 vaults is safe only as long as they're not empty or can be emptied. This is the case for many vaults out there that represent already established DeFi products. Those vaults, despite not being allowed by default and after a sufficient audit and testing, could in principle be safely allowed to be used as strategies in EulerEarn.

Fix Review: No fix, acknowledged by Euler.

Informational Issues

I-01. Cache `lastTotalAssets` to Save Gas on Multiple Storage Reads

Location: eulerearn.sol/#L901-L906

Description:
The `_accruedFeeAndAssets()` function reads `lastTotalAssets` from storage multiple times, causing unnecessary SLOAD operations. Multiple reads of `lastTotalAssets` waste gas on redundant SLOAD operations.

Recommendation: Cache `lastTotalAssets` to a local variable.

Fix Review: Fix confirmed.

Formal Verification

Verification Methodology

We performed verification of the EulerEarn protocol using the Certora verification tool, which is based on Satisfiability Modulo Theories (SMT). In short, the Certora verification tool works by compiling formal specifications written in the Certora Verification Language (CVL) and EulerLabs' implementation source code written in Solidity.
More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology, it means the specification in CVL holds for all possible inputs. However, specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g., to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed, and assertions that are verified on any resulting states from the computation.

Inductive Invariants: Inductive invariants are proved by induction on the structure of a smart contract. We use constructors as a base case, and consider all other (relevant) externally callable functions that can change the storage as step cases.
Specifically, to prove the base case, we show that a property holds in any resulting state after a symbolic call to the respective constructor. For proving step cases, we generally assume a state where the invariant holds (induction hypothesis), symbolically execute the functionality under investigation, and prove that after this computation, any resulting state satisfies the invariant.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

1. We work with contracts inherited from the original contracts that we call harnesses. In the inherited contracts, we add more view methods, flags, etc. In cases where it was not possible to collect the required information via the inherited object.
2. We replaced some functions with equivalent CVL implementations. Notably, *mulDiv, safeTransfer, and safeTransferFrom*. This speeds up the verification process and doesn't affect the results.
3. We assume that loops are not iterated through more than two times.
4. We use mocks to resolve the behavior of unknown contracts that will be used by the main contract. In this case, we specify the allowed ERC4626 behavior in ERC4626.spec. Additionally, we use the PerspectiveMock.sol as a mock for the perspective contract.

Formal Verification Properties Overview

ID	Title	Impact	Status
P-01	Consistency	Consistency properties of the contract state	Verified
P-02	Range	Contract variables stay within allowed ranges	Verified
P-03	PendingValues	Pending values are consistent	Verified
P-04	Roles	All possible reverts respect role hierarchy	Verified
P-05	Timelock	Timelock behaves correctly	Verified
P-06	Enabled	Consistency between enabled markets in supply and withdraw queues.	Verified
P-07	Liveness	In emergency situations allocator can pause supply and curator can remove a market	Verified
P-08	Immutability	No Delegate Calls	Verified
P-09	Reentrancy	All external calls are from known sources.	Verified
P-10	Reverts	Methods revert on incorrect inputs and do not revert otherwise.	Verified
P-11	Balances	Methods update balances correctly.	Verified
P-12	Assets	totalAssets, lastTotalAssets, and lostAssets are consistent	Verified

Formal Verification Properties

Module General Assumptions

We introduced two more mappings and a constant to the EulerEarn contract:

- mapping(address => uint256) public withdrawRank
- mapping(address => uint256) public deletedAt
- uint256 lastIndexWithdraw.

We also modified some of its methods to correctly maintain these. These changes don't affect the original functionality of the contract and help us to verify rules about the withdrawQueue.

Module Properties

P-01. Reachable states are consistent	
Status: Verified	Rules from ConsistentState.spec

Rule Name	Status	Description	Link to rule report
noFeeToUnsetFeeRecipient	Verified	*If feeRecepient is not set, then fee() returns zero.*	rule report
supplyCapIsEnabled	Verified	*If the market has a cap greater than 0, then it is enabled.*	
pendingSupplyCapHasConsistentAsset	Verified	*If the market has a pending cap, then its token is the same as the asset of the vault.*	
enabledHasConsistentAsset	Verified	*If the market is enabled, then its token is the same as the asset of the vault.*	
supplyCapIsNotMarkedForRemoval	Verified	*If the market has a non-zero supply cap, then it's not marked for removal. (I.e., removableAt == 0}*	

notEnabledIsNotMarkedForRemoval	Verified	*If the market is enabled, then it's not marked for removal. (I.e., removableAt == 0}*	
pendingCapIsNotMarkedForRemoval	Verified	*If the market has a pending cap then it's not marked for removal. (I.e., removableAt == 0}*	
newSupplyQueueEnsuresPositiveCap	Verified	*Method SetSupplyQueue can only add markets with non-zero caps.*	
noCapThenNoApproval	Verified	*If a market has zero cap, then EulerEarn does not have approval of the asset token for it.*	
notInWithdrawQThenNoApproval	Verified	*If a market is not in the withdraw queue, then EulerEarn does not have approval of the asset token for it.*	

P-02. Contract variables stay within allowed ranges

Status: Verified	Rules from Range.spec

Rule Name	Status	Description	Link to rule report
pendingTimelockInRange	Verified	*pendingTimelock_ is within minTimelock and maxTimelock at all times.*	rule report
timelockInRange	Verified	*timelock is within minTimelock and maxTimelock at all times.*	
feeInRange	Verified	*fee is less than maxFee at all times.*	
supplyQueueLengthInRange	Verified	*The length of SupplyQueue is less than maxQueueLength at all times.*	

withdrawQueueLengthInRange	Verified	*The length of WithdrawQueue is less than maxQueueLength at all times.*	
pendingCapIsUint184	Verified	*pendingCap.value is never larger than 2^184.*	

P-03. Pending values are consistent

Status: Verified	Rules from PendingValues.spec

Rule Name	Status	Description	Link to rule report
noBadPendingTimelock	Verified	*pendingTimelock.validAt is zero if and only if the pendingTimelock value is zero.*	rule report
smallerPendingTimelock	Verified	*The pending timelock value is always strictly smaller than the current timelock value.*	
greaterPendingCap	Verified	*The pending cap value is either 0 or strictly greater than the current cap value.*	
differentPendingGuardian	Verified	*The pending guardian is either the zero address or it is different from the current guardian.*	

P-04. Roles hierarchy

Status: Verified	Rules from Roles.spec

Rule Name	Status	Description	Link to rule report
ownerIsGuardian	Verified	*If the Guardian can perform an action then the Owner can also perform it.*	rule report
ownerIsCurator	Verified	*If the Curator can perform an action then the Owner can also perform it.*	
curatorIsAllocator	Verified	*If the Allocator can perform an action then the Curator can also perform it.*	

P-05. Timelocks work correctly

Status: Verified	Rules from Timelock.spec

Rule Name	Status	Description	Link to rule report
guardianUpdateTime	Verified	*No change of guardian can happen before the timelock.*	rule report
capIncreaseTime	Verified	*No increase of cap can happen before the timelock.*	
timelockDecreaseTime	Verified	*To decrease the timelock you must wait for the current timelock to pass.*	

removableTime	Verified	*The nextRemovableTime is increasing with time and no removal can happen before it.*	
nextGuardianUpdateTimeDoesNotRevert	Verified	*No reverts are introduced due to auxiliary functions nextGuardianTime, nextCapIncreaseTime, nextTimelockDecreaseTime and nextRemovableTime.*	
nextCapIncreaseTimeDoesNotRevert	Verified		
nextTimelockDecreaseTimeDoesNotRevert	Verified		
nextRemovableTimeDoesNotRevert	Verified		

P-06. Consistency of Supply and Withdraw queues

Status: Verified	Rules from Enabled.spec

Rule Name	Status	Description	Link to rule report
distinctIdentifiers	Verified	*There are no duplicate markets in the withdrawQueue.*	rule report
inWithdrawQueueIsEnabled	Verified	*If the market is in the WithdrawQueue then it is enabled.*	
inWithdrawQueueIsEnabledPreservedUpdateWithdrawQueue	Verified	*If the market is in the WithdrawQueue then it is enabled – the case of updateWithdrawQueue requires separate proof.*	
withdrawRankCorrect	Verified	*Auxiliary mapping withdrawRank holds the index values of the markets in the withdraw queue.*	

enabledHasPositiveRank	Verified	*All enabled markets have a positive withdrawRank.*	
nonZeroCapHasPositiveRank	Verified	*If the market has a non-zero cap, then it's in the WithdrawQueue.*	
setSupplyQueueRevertsOnInvalidInput	Verified	*The method setSupplyQueue reverts if one of the markets would have zero cap.*	
addedToSupplyQThenIsInWithdrawQ	Verified	*If a market is added to the supplyQueue then it is present in the withdrawQueue.*	
enabledIsInWithdrawalQueue	Verified	*If the market is enabled, then it's in the WithdrawQueue.*	

P-07. Emergency Handling

Status: Verified	Rules from Liveness.spec

Rule Name	Status	Description	Link to rule report
canPauseSupply	Verified	*The allocator is able to pause supply by setting an empty supplyQueue. After the pause, all deposits and mints revert.*	rule report
canForceRemoveMarket	Verified	*The curator role is able to remove any market.*	

P-08. Delegate Calls Safety

Status: Verified	Rules from Immutability.spec

Rule Name	Status	Description	Link to rule report
noDelegateCalls	Verified	*No delegateCall happens, i.e., the contract is truly immutable.*	rule report

P-09. Reentrancy Safety

Status: Verified	Rules from Reentrancy.spec

Rule Name	Status	Description	Link to rule report
reentrancySafe	Verified	*All external calls come from specific, known functions marked as trusted in the methods block.*	rule report

P-10. Methods revert on incorrect inputs and don't revert otherwise

Status: Verified	Rules from Reverts.spec

Rule Name	Status	Description	Link to rule report
setCuratorRevertCondition	Verified	*setCurator reverts if and only if the specified conditions occur.*	rule report

setIsAllocatorRevertCondition	Verified	*setAllocator reverts if and only if the specified conditions occur.*
setFeeInputValidation	Verified	*setFee reverts if the specified conditions occur.*
setFeeRecipientInputValidation	Verified	*setFeeRecipient reverts if the specified conditions occur.*
submitGuardianRevertCondition	Verified	*submitGuardian reverts if and only if the specified conditions occur.*
submitCapRevertCondition	Verified	*submitCap reverts if and only if the specified conditions occur.*
submitMarketRemovalRevertCondition	Verified	*submitMarketRemoval reverts if and only if the specified conditions occur.*
setSupplyQueueInputValidation	Verified	*setSupplyQueue reverts if the specified conditions occur.*
updateWithdrawQueueInputValidation	Verified	*updateWithdrawQueue reverts if the specified conditions occur.*
reallocateInputValidation	Verified	*reallocate reverts if the specified conditions occur.*
revokePendingTimelockRevertCondition	Verified	*revokePendingTimelock reverts if and only if the specified conditions occur.*
revokePendingGuardianRevertCondition	Verified	*revokePendingGuardian reverts if and only if the specified conditions occur.*
revokePendingCapRevertCondition	Verified	*revokePendingCap reverts if and only if the specified conditions occur.*
revokePendingCapRevertCondition	Verified	*revokePendingCap reverts if and only if the specified conditions occur.*

revokePendingMarketRemovalRevertCondition	Verified	*revokePendingMarketRemoval reverts if and only if the specified conditions occur.*	
acceptTimelockRevertCondition	Verified	*acceptTimelock reverts if and only if the specified conditions occur.*	
acceptGuardianRevertCondition	Verified	*acceptGuardian reverts if and only if the specified conditions occur.*	
acceptCapInputValidation	Verified	*acceptCap reverts if the specified conditions occur.*	

P-11. Methods update balances correctly

Status: Verified	Rules from Balances.spec

Rule Name	Status	Description	Link to rule report
depositTokenChange	*Verified*	*Depositing correctly updates balances of all involved parties.*	rule report
withdrawTokenChange	*Verified*	*Withdrawing correctly updates balances of all involved parties.*	
reallocateTokenChange	*Verified*	*Calling reallocate doesn't affect balances of SiloVault, msg.sender or any market.*	
vaultBalanceNeutral	*Verified*	*Vault sends out all tokens that it receives.*	
onlySpecicifiedMethodsCanDecreaseMarketBalance	*Verified*	*Only withdraw, redeem and reallocate can decrease the balance of EulerEarn in its asset.*	

P-12. Last Total Assets and Lost Assets are consistent

Status: Verified	Rules from LostAssets.spec

Rule Name	Status	Description	Link to rule report
lostAssetsIncreases	*Verified*	*The value of lostAssets always increases (or does not change)*	rule report
lastTotalAssetsSmallerThanTotalAssets	*Verified*	*lastTotalAssets() always holds a value that is at most the value of totalAssets()*	
lastTotalAssetsIncreases	*Verified*	*lastTotalAssets() increases (or is unchanged) in all operations except for withdraw, redeem and updateWithdrawQueue, and decreases appropriately on withdraw and redeem*	
lastTotalAssetsDecreasesCorrectlyOnWithdraw	*Verified*		
lastTotalAssetsDecreasesCorrectlyOnRedeem	*Verified*		
sharePriceIncreases	*Verified*	*The price of a single share of eulerEarn can only increase*	

Disclaimer

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.