certora

Security Assessment & Formal Verification Final Report

Euler Vault Kit

August 2024

Prepared for
Euler

certora

Euler

Table of content

Project Summary

Project Scope

Project Name	Repository	Last Commit Hash	Platform
Euler Vault Kit	https://github.com/euler-xyz/euler-vault-kit	d674d7501853643796081591d686c64d811ad1c1	EVM/Solidity 0.8.23

Project Overview

This document describes the findings of the Euler Vault Kit project using the Certora Prover and manual code review. The work was undertaken from March 7th 2024 to May 8th 2024.

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 Certora team performed a manual audit of all the Solidity contracts in addition to writing formal rules. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed below.

Protocol Overview

The Euler Vault Kit is a system for constructing credit vaults. Credit vaults are ERC-4626 vaults with added borrowing functionality. Unlike typical ERC-4626 vaults which earn yield by actively investing deposited funds, credit vaults are passive lending pools. See the whitepaper for more details.

Users can borrow from a credit vault as long as they have sufficient collateral deposited in other credit vaults. The liability vault (the one that was borrowed from) decides which credit vaults are acceptable as collateral. Interest is charged to borrowers by continuously increasing the amount of their outstanding liability and this interest results in yield for the depositors.

Vaults are integrated with the Ethereum Vault Connector contract (EVC), which keeps track of the vaults used as collateral by each account. In the event a liquidation is necessary, the EVC allows a liability vault to withdraw collateral on a user's behalf.

The EVC is also an alternate entry-point for interacting with vaults. It provides multicall-like batching, simulations, gasless transactions, and flash liquidity for efficient refinancing of loans. External contracts can be invoked without needing special adaptors, and all functionality is accessible to both EOAs and contract wallets. Although each address is only allowed one outstanding liability at any given time, the EVC provides it with 256 virtual addresses, called sub-accounts (from here on, just accounts). Sub-account addresses are internal to the EVC and compatible vaults, and care should be taken to ensure that these addresses are not used by other contracts.

The EVC is responsible for authentication, and vaults are responsible for authorisation. For example, if a user attempts to redeem a certain amount, the EVC makes sure the request actually came from the user, and the vault makes sure the user actually has this amount.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	0	0	0
Medium	8	5	5
Low	3	3	3
Total	11	8	8

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Probability		

Detailed Findings

ID	Title	Severity	Status
M-01	Griefing bad debt socialization	Medium	Won't fix
M-02	newTotalBorrows can revert due to overflow	Medium	Fixed
M-03	nonReentrant modifier implementation will probably result in a DOS of future vaults inheriting from BaseProductLine	Medium	Fixed
M-04	ConfigAmount.mul can overflow	Medium	Fixed
M-05	Repayments Paused While Liquidations Enabled	Medium	Out of Scope (Governance's responsibility)
M-06	After unpausing repay and liquidation, the users are immediately liquidatable	Medium	Out of Scope (Governance's responsibility)
M-07	Lacking the feature to transfer the admin role in ProtocolConfig	Medium	Fixed
M-08	Fallback lacking payable and callThroughEVC-enabled functions lacking payable	Medium	Fixed
L-01	Lack of compliance from the ERC4626 standard	Low	Fixed

L-02	Self-transfer would break accounting if the path was opened	Low	Fixed
L-03	Variables should either be made immutable or have setters	Low	Acknowledged but no longer relevant

Medium Severity Issues

M-01. Griefing bad debt socialization

In this liquidation scenario, there are 2 collaterals enabled.

Say a liquidator calls liquidate on the 1st collateral, leaving 0 of it in the violator's balance.

Now, in a second transaction, the liquidator calls liquidate on the 2nd collateral. What is expected is that if there are debts left that are higher than the amount repaid, then this debt would be socialized.

However, it's possible for an attacker to frontrun this second liquidation by sending 1 wei of the first collateral to the violator midway. Due to this 1 wei, the call to `checkNoCollateral()` will return false and debt socialization won't happen:

- Liquidation.sol#L201-L206

```
File: Liquidation.sol
201:         // Handle debt socialization
202:
203:         if (
204:             vaultCache.configFlags.isNotSet(CFG_DONT_SOCIALIZE_DEBT) &&
liqCache.liability > liqCache.repay
205:                 && checkNoCollateral(liqCache.violator,
liqCache.collaterals)
206:         ) {
```

- LiquidityUtils.sol#L56-L71

```
File: LiquidityUtils.sol
56:     // Check if the account has no collateral balance left, used for debt
socialization
57:     // If LTV is zero, the collateral can still be liquidated.
58:     // If the price of collateral is zero, liquidations are not executed,
so the check won't be performed.
59:     // If there is no collateral balance at all, then debt socialization
```

```
can happen.
60:     function checkNoCollateral(address account, address[] memory
collaterals) internal view virtual returns (bool) {
61:         for (uint256 i; i < collaterals.length; ++i) {
62:             address collateral = collaterals[i];
63:
64:             if (!isRecognizedCollateral(collateral)) continue;
65:
66:             uint256 balance = IERC20(collateral).balanceOf(account);
67:             if (balance > 0) return false;
68:         }
69:
70:         return true;
71:     }
```

Given the dust collateral balance on the violator, there's no incentive for liquidators to act, hence forcing a good-willed actor to act at a loss to liquidate this final 1 wei (which could be expensive on mainnet).

Thankfully, a batch operation from the EVC could fully liquidate a violator, but would still cost quite a bit of gas (`liquidate()` is arguably more gas-expensive than a call to `transfer` on an Escrow Vault).

Euler's response: It's unclear how much of a concern it should be. One would imagine that there are beneficiaries of the pool that would want to see bad debt socialized and will pay to do it even if there's 1 Wei collateral leftover. In practice I fully expect lenders will themselves become liquidators. When they liquidate a pool they will: withdraw, liquidate, socialize bad debt, re-deposit. And it will be competitive for this reason. No one will want to be the one having bad debt pushed on them.
In addition, liquidators can use a smart contract that loops over each of the collateral and liquidates it. This will prevent the front-running attack described, and the gas cost should be minimal because:

* The number of collaterals are limited by the EVC (and optionally limited further by vaults)
* Much of the storage costs are amortised
* Zeroing out dust should result in gas refunds

M-02. `newTotalBorrows` can revert due to overflow

In this violation from the prover, the following can revert:

- Cache.sol#L87-L88

```
File: Cache.sol
91:             uint256 newTotalBorrows =
92:                 vaultCache.totalBorrows.toUint() * newInterestAccumulator
/ vaultCache.interestAccumulator;
```

Indeed, while `vaultCache.totalBorrows` is at most `MAX_UINT144`, `newInterestAccumulator` can be closer to `MAX_UINT256` by a margin of `1e27` and even have a close to overflowable value as can be seen here:

- Cache.sol#L76-L85

```
File: Cache.sol
79:             unchecked {
80:                 (uint256 multiplier, bool overflow) =
RPow.rpow(interestRate + 1e27, deltaT, 1e27);
81:
82:                 // if exponentiation or accumulator update overflows, keep
the old accumulator
83:                 if (!overflow) {
84:                     uint256 intermediate = newInterestAccumulator *
multiplier;
85:                     if (newInterestAccumulator == intermediate /
multiplier) {
86:                         newInterestAccumulator = intermediate / 1e27;
87:                     }
88:                 }
89:             }
```

As we're in the deep in the `initOperation` function which is called almost everywhere, this means that the overflow could result in a permanent Denial of Service, while it is instead expected to be working with the "old accumulator"

Euler's response: We acknowledge this. It should be fixed in the following PR: https://github.com/euler-xyz/euler-vault-kit/pull/184

M-03. nonReentrant modifier implementation will probably result in a DOS of future vaults inheriting from BaseProductLine

If we look here:

```
File: BaseProductLine.sol
40:     modifier nonReentrant() {
41:         if (reentrancyLock != REENTRANCYLOCK__UNLOCKED) revert
E_Reentrancy();
42:
43:         reentrancyLock = REENTRANCYLOCK__LOCKED;
44:         _;
45:         reentrancyLock = REENTRANCYLOCK__UNLOCKED;
46:     }
```

The `if (reentrancyLock != REENTRANCYLOCK__UNLOCKED) revert E_Reentrancy();` part forces the dev to remember implementing `reentrancyLock = REENTRANCYLOCK__LOCKED;` somewhere in the code in the future.

If we look at `BaseProductLine` and its child contracts (`Core` and `Escrow`): `nonReentrant` is never used, so it's fine for now.

But, the next developer on another contract will need to not forget to set in the `constructor` the `reentrancyLock = REENTRANCYLOCK__LOCKED` code. This can be considered not developer-friendly and is a bit of a risk.

The implementation in `GenericFactory` didn't forget to set the value in the `constructor`, thankfully.

However, there exists another implementation in the codebase that would incidentally set `reentrancyLock` for the first time if it wasn't done in the `constructor`:

```
File: EulerSavingsRate.sol
42:     modifier nonReentrant() {
43:         if (esrSlot.locked == REENTRANCYLOCK__LOCKED) revert Reentrancy();
44:
45:         esrSlot.locked = REENTRANCYLOCK__LOCKED;
```

```
46:         _;
47:         esrSlot.locked = REENTRANCYLOCK__UNLOCKED;
48:     }
```

Both implementations exist in the codebase (checking `if == locked` or `if != unlocked`).

Consider standardizing the code using the more maintainable version, which also follows OpenZeppelin's pattern for `ReentrancyGuard`:
At https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/utils/ReentrancyGuardUpgradeable.sol#L82-L84, it can be seen that the `revert` is on `if ($._status == ENTERED)` and then sets `$._status = ENTERED` if the code passes. They're not reverting on if `($._status != NOT_ENTERED)` which would be similar to `BaseProductLine` and `GenericFactory`'s implementation.
This also automatically avoids an unintentional DOS of the system in case a developer forgot to call `__ReentrancyGuard_init()`

```
ReentrancyGuardUpgradeable.sol
        if ($._status == ENTERED) {
            revert ReentrancyGuardReentrantCall();
        }
```

Euler's response: We acknowledge this was a legitimate finding. However, we have decided to remove the Product Lines contracts altogether, so this is no longer relevant.

M-04. `ConfigAmount.mul` can overflow

In ConfigAmount.sol#L20-L25:

```
File: ConfigAmount.sol
20:     // note assuming arithmetic checks are already performed
21:     function mul(ConfigAmount self, uint256 multiplier) internal pure
returns (uint256) {
22:         unchecked {
23:             return uint256(self.toUint16()) * multiplier / 1e4;
24:         }
25:     }
```

There's an unchecked statement in which we multiply between a `collateralValue` and `ltv`.

As, technically, `collateralValue` can be close to `type(uint256).max`, this can overflow.

As this function is only called once in the codebase at LiquidityUtils.sol#L100:

```
        return ltv.mul(currentCollateralValue);
```

and as there isn't an explicit check preventing overflows: the comment assuming that "arithmetic checks are already performed" is not true.

If we happen to have some unexpected combination of `unitOfAccount` and `collateral`, the overflow could happen (this is still very unlikely as `unitOfAccount` isn't user-controlled and collaterals are to be added by the Governance).

Euler's response: I think you're right, this could overflow, and the comment is wrong, or at least, the code using the function is not respecting it. Good find!
As for the fix, I think removing the unchecked block should be enough. The collateral value would need to be > type(uint256).max / 1e4 which shouldn't really happen in a legitimate scenario. If the attacker can push it this high, then probably they can push it just a little bit further and there's nothing to do. On the other hand we could be checking for overflow and dividing by 1e4 first if detected, which would be an easy fix.
We're now considering the simplest solution, and while we're at it, cleaning the ConfigAmount lib a little bit:
https://github.com/euler-xyz/euler-vault-kit/pull/122
A huge collateral value which could cause a problem means we're at the numerical limits, where overflows are expected. It can't really happen if the vault is properly configured unless a manipulation is taking place, in which case the vault is compromised anyway.

M-05. Repayments Paused While Liquidations Enabled

Repayments being paused but liquidations being enabled would unfairly prevent Borrowers from making their repayments while still allowing them to be liquidated.
It seems here that repayments can be paused while liquidations can remain enabled:

```
File: Liquidation.sol
44:     function liquidate(address violator, address collateral, uint256
```

```
repayAssets, uint256 minYieldBalance)
45:         public
46:         virtual
47:         nonReentrant
48:     {
49:         (MarketCache memory marketCache, address liquidator) =
initOperation(OP_LIQUIDATE, CHECKACCOUNT_CALLER);  //@audit-issue no parallel
checks for OP_REPAY
+ 49:          if (marketCache.disabledOps.check(OP_REPAY)) {
+ 49:             revert E_OperationDisabled();
+ 49:         } //<------- recommended additional check
```

Euler's response: We acknowledge this as a legitimate concern. However, in our view, the EVK is only responsible for providing the mechanism for pausing, and enforcing fair policies should be up to the governor and/or the hook policies it installs. To enforce this on-chain, the governor should be a contract with limited hook modification abilities. The Governance contracts will be the ones in charge of fairly implementing the pause/unpause mechanisms

M-06. After unpausing repay and liquidation, the users are immediately liquidatable

There should be a grace period allowing users to repay what they owe.

Euler's response: Acknowledged but out of scope for the EVK considered as a general-purpose vault kit. The Governance contracts will be the ones in charge of fairly implementing the pause/unpause mechanisms

M-07. Lacking the feature to transfer the `admin` role in `ProtocolConfig`

The `admin` variable is set in the constructor but can't be changed afterwards

Consider adding an `onlyAdmin` function to the `ProtocolConfig` to enable transferring the `admin` role.

Euler's response: Fixed in #59

M-08. Fallback lacking `payable` and `callThroughEVC`-enabled functions lacking `payable`

When a function isn't marked as payable, there's a silent check that `msg.value == 0` (which costs some additional gas compared to payable functions).

Given that a `BeaconProxy` contains an upgradeable EVault's state and that `callThroughEVCInternal()` can expect an `msg.value`, it feels like some `payable` keywords are missing, like on the fallback itself:

- src/GenericFactory/BeaconProxy.sol

```
# File: src/GenericFactory/BeaconProxy.sol

BeaconProxy.sol:45:     fallback() external {
```

And also on all functions containing the `callThroughEVC` modifier:

```
File: Dispatch.sol
73:     modifier callThroughEVC() {
74:         if (msg.sender == address(evc)) {
75:             _;
76:         } else {
77:             callThroughEVCInternal();
78:         }
79:     }
...
123:     function callThroughEVCInternal() private {
...
131:             mstore(68, callvalue()) // EVC.call 3rd argument - msg.value
```

```
src/EVault/EVault.sol:
   32:     function transfer(address to, uint256 amount) public override
virtual callThroughEVC returns (bool) { return super.transfer(to, amount); }
   34:     function transferFrom(address from, address to, uint256 amount)
public override virtual callThroughEVC returns (bool) { return
super.transferFrom(from, to, amount); }
   38:     function transferFromMax(address from, address to) public override
```

```
virtual callThroughEVC returns (bool) { return super.transferFromMax(from,
to); }
   75:     function deposit(uint256 amount, address receiver) public override
virtual callThroughEVC returns (uint256) { return super.deposit(amount,
receiver); }
   77:     function mint(uint256 amount, address receiver) public override
virtual callThroughEVC use(MODULE_VAULT) returns (uint256) {}
   79:     function withdraw(uint256 amount, address receiver, address owner)
public override virtual callThroughEVC returns (uint256) { return
super.withdraw(amount, receiver, owner); }
   81:     function redeem(uint256 amount, address receiver, address owner)
public override virtual callThroughEVC use(MODULE_VAULT) returns (uint256) {}
   83:     function skim(uint256 amount, address receiver) public override
virtual callThroughEVC use(MODULE_VAULT) returns (uint256) {}
  108:     function borrow(uint256 amount, address receiver) public override
virtual callThroughEVC use(MODULE_BORROWING) returns (uint256) {}
  110:     function repay(uint256 amount, address receiver) public override
virtual callThroughEVC use(MODULE_BORROWING) returns (uint256) {}
  112:     function loop(uint256 amount, address sharesReceiver) public
override virtual callThroughEVC use(MODULE_BORROWING) returns (uint256) {}
  114:     function deloop(uint256 amount, address debtFrom) public override
virtual callThroughEVC use(MODULE_BORROWING) returns (uint256) {}
  116:     function pullDebt(uint256 amount, address from) public override
virtual callThroughEVC use(MODULE_BORROWING) returns (uint256) {}
  120:     function touch() public override virtual callThroughEVC
use(MODULE_BORROWING) {}
  128:     function liquidate(address violator, address collateral, uint256
repayAssets, uint256 minYieldBalance) public override virtual callThroughEVC
use(MODULE_LIQUIDATION) {}
  199:     function convertFees() public override virtual callThroughEVC
use(MODULE_GOVERNANCE) {}

src/Synths/ESVault.sol:
  62:     function deposit(uint256 amount, address receiver) public virtual
override callThroughEVC returns (uint256) {
```

Euler's response: Fixed in #64

Low Severity Issues

L-01. Lack of compliance from the ERC4626 standard

From https://eips.ethereum.org/EIPS/eip-4626

1. The `nonReentrantView` makes several "MUST NOT REVERT" functions to revert
2. The pause mechanism makes several "MUST NOT REVERT" functions to revert

Euler's response:
The whitepaper was updated.

L-02. Self-transfer would break accounting if the path was opened

Note: This is informational because the current code-path isn't opened. However, the fix is simple enough so that, in case of a future decision of making self-transfers possible, then the vulnerability would be known/fixed

The internal `transferBorrow()` function is implemented using a dangerous pattern. If, one day, a codepath is opened to call it with `from == to`, then `to`'s balance will be increased by `amount`.

While the codepath isn't opened due to high level checks, a future dev mistake could be made in the protocol's future.

Moving the `from != to` isn't the recommended solution here as we can simply rearrange the sequence of calls like this:

```
File: BorrowUtils.sol
- 71:          (Owed toOwed, Owed toOwedPrev) = updateUserBorrow(vaultCache,
to);
...
- 85:          toOwed = toOwed + amount;
86:
87:         vaultStorage.users[from].setOwed(fromOwed);
+ 87:          (Owed toOwed, Owed toOwedPrev) = updateUserBorrow(vaultCache,
to);
+ 87:          toOwed = toOwed + amount;
88:         vaultStorage.users[to].setOwed(toOwed);
```

And the accounting would be ok without any additional gas cost.

The exact same pattern exists for `transferBalance()`

Euler's response: We intend to fix this in the following PR:
https://github.com/euler-xyz/euler-vault-kit/pull/181

L-03. Variables should either be made immutable or have setters

These 2 variables are only set in the constructor and nowhere else after: ProductLines/Core.sol#L15-L16 (not even in the inheritance). While they could be made immutable to save a lot of gas, I feel that here, it's more likely that `governorOnly` setters are lacking

Euler's response: Acknowledged, but Product Line contracts are being removed so this is no longer relevant.

Informational Severity Issues

I-01. Using both `getQuotes` and `getQuote` in `liquidate()` is confusing

To calculate if a user is a violator who can be liquidated, `getQuotes()` is called (so, the bid price for collateral is used, and the ask price for liability is used).
However, afterwards, `getQuote()` is used (so, the mid-point price is used for both the collateral and liability value to compute the amounts to repay and the bonus earned by the liquidator)

This is a bit confusing.

Euler's response: We've decided to make liquidation use mid-point prices for everything. See #124

I-02. The Base contract doesn't need to be inherited as often

The inheritance instructions can be tidied up by removing the ones being already inherited by the parents

```
File: Initialize.sol
- 18: abstract contract InitializeModule is IInitialize, Base, BorrowUtils {
+ 18: abstract contract InitializeModule is IInitialize, BorrowUtils {

File: Borrowing.sol
- 19: abstract contract BorrowingModule is IBorrowing, Base, AssetTransfers,
BalanceUtils, LiquidityUtils {
+ 19: abstract contract BorrowingModule is IBorrowing, AssetTransfers,
BalanceUtils, LiquidityUtils {

File: Governance.sol
- 19: abstract contract GovernanceModule is IGovernance, Base, BalanceUtils,
BorrowUtils, LTVUtils {
+ 19: abstract contract GovernanceModule is IGovernance, BalanceUtils,
BorrowUtils, LTVUtils {

...and others...
```

Euler's response: Fixed in: #226

I-03. BPS_SCALE's format is questionable

While it's understandable that this would mean 100%, this is actually not something usually seen and is therefore a bit confusing.

BPS is a known constant being used across numerous protocols so it doesn't need that special format.

```
File: PegStabilityModule.sol
- 13:     uint256 public constant BPS_SCALE = 100_00;
+ 13:     uint256 public constant BPS_SCALE = 10_000;
```

Euler's response: Acknowledged, will keep as is. This format was suggested by other auditor.

I-04. 60000 as a scale is confusing

Using BPS (10_000) is usually the standard

- ConfigAmount.sol#L8-L12

```
// ConfigAmounts are floating point values encoded in 16 bits with a
CONFIG_SCALE precision (60 000).
// The type is used to store protocol configuration values.

library ConfigAmountLib {
    uint256 constant CONFIG_SCALE = 60_000; // fits in uint16
```

Euler's response: Fixed in #20

I-05. Default Visibility for constants

Some constants are using the default visibility. For readability, consider explicitly declaring them as `internal`.

Affected code:

- src/EVault/shared/types/Snapshot.sol

```
# File: src/EVault/shared/types/Snapshot.sol

Snapshot.sol:15:     uint8 constant STAMP = 1;  // non zero initial value of
the snapshot slot to save gas on SSTORE
```

- src/EVault/shared/types/UserStorage.sol

```
# File: src/EVault/shared/types/UserStorage.sol

UserStorage.sol:18:     uint256 constant BALANCE_FORWARDER_MASK =
0x8000000000000000000000000000000000000000000000000000000000000000;

UserStorage.sol:19:     uint256 constant OWED_MASK =
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000000000000;

UserStorage.sol:20:     uint256 constant SHARES_MASK =
0x000000000000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFFFFF;

UserStorage.sol:21:     uint256 constant OWED_OFFSET = 112;
```

- src/GenericFactory/BeaconProxy.sol

```
# File: src/GenericFactory/BeaconProxy.sol

BeaconProxy.sol:7:     bytes32 constant BEACON_SLOT =
0xa3f0ad74e5423aebfd80d3ef4346578335a9a72aeaee59ff6cb3582b35133d50;

BeaconProxy.sol:9:     bytes32 constant IMPLEMENTATION_SELECTOR =
0x5c60da1b00000000000000000000000000000000000000000000000000000000;

BeaconProxy.sol:11:     uint256 constant MAX_TRAILING_DATA_LENGTH = 128;
```

- src/GenericFactory/GenericFactory.sol

```
# File: src/GenericFactory/GenericFactory.sol
```

```
GenericFactory.sol:15:     uint256 constant REENTRANCYLOCK__UNLOCKED = 1;

GenericFactory.sol:16:     uint256 constant REENTRANCYLOCK__LOCKED = 2;
```

- src/ProtocolConfig/ProtocolConfig.sol

```
# File: src/ProtocolConfig/ProtocolConfig.sol

ProtocolConfig.sol:9:     uint256 constant MIN_INTEREST_FEE = CONFIG_SCALE * 1
/ 100; // 1%

ProtocolConfig.sol:10:     uint256 constant MAX_INTEREST_FEE = CONFIG_SCALE *
50 / 100; // 50%

ProtocolConfig.sol:11:     uint256 constant PROTOCOL_FEE_SHARE = 0.1 * 1e18;
```

- src/interestRateModels/BaseIRM.sol

```
# File: src/interestRateModels/BaseIRM.sol

BaseIRM.sol:10:     uint256 constant MAX_ALLOWED_INTEREST_RATE = uint256(5 *
1e27) / SECONDS_PER_YEAR; // 500% APR
```

- src/interestRateModels/IRMClassLido.sol

```
# File: src/interestRateModels/IRMClassLido.sol

IRMClassLido.sol:19:     uint256 constant SECONDS_PER_DAY = 24 * 60 * 60;

IRMClassLido.sol:20:     uint256 constant MAX_ALLOWED_LIDO_INTEREST_RATE =
1e27 / SECONDS_PER_YEAR; // 100% APR

IRMClassLido.sol:21:     uint256 constant LIDO_BASIS_POINT = 10000;
```

Euler's response: Fixed in #227

UserStorage.sol, BeaconProxy.sol, GenericFactory.sol: Fixed here
ProtocolConfig.sol: constants mentioned removed
BaseIRM.sol, IRMClassLido.sol: contracts removed

I-06. Change uint to uint256

Throughout the code base, some variables are declared as `uint`. To favor explicitness, consider changing all instances of `uint` to `uint256`

Affected code:

- src/EVault/modules/Vault.sol

```
# File: src/EVault/modules/Vault.sol

Vault.sol:270:          uint remainingSupply;
```

- src/EVault/shared/LiquidityUtils.sol

```
# File: src/EVault/shared/LiquidityUtils.sol

LiquidityUtils.sol:41:          uint collateralValue;

LiquidityUtils.sol:72:      function getLiabilityValue(MarketCache memory
marketCache, address account, Owed owed) internal view returns (uint value) {

LiquidityUtils.sol:84:      function getCollateralValue(MarketCache memory
marketCache, address account, address collateral, LTVType ltvType) internal
view returns (uint value) {
```

Euler's response: All of these instances have been fixed.

I-07. Use Underscores for Number Literals (add an underscore every 3 digits)

Affected code:

- src/EVault/shared/Constants.sol

```
# File: src/EVault/shared/Constants.sol

Constants.sol:10: uint256 constant SECONDS_PER_YEAR = 365.2425 * 86400; //
Gregorian calendar
```

- src/interestRateModels/BaseIRM.sol

```
# File: src/interestRateModels/BaseIRM.sol

BaseIRM.sol:8:     uint256 internal constant SECONDS_PER_YEAR = 365.2425 *
86400; // Gregorian calendar
```

- src/interestRateModels/IRMClassLido.sol

```
# File: src/interestRateModels/IRMClassLido.sol

IRMClassLido.sol:21:     uint256 constant LIDO_BASIS_POINT = 10000;

IRMClassLido.sol:38:         slope1 = 709783723;

IRMClassLido.sol:39:         slope2 = 37689273223;

IRMClassLido.sol:40:         kink = 3435973836;
```

- src/interestRateModels/IRMClassMajor.sol

```
# File: src/interestRateModels/IRMClassMajor.sol

IRMClassMajor.sol:12:             1681485479,

IRMClassMajor.sol:13:             44415215206,

IRMClassMajor.sol:14:             3435973836
```

- src/interestRateModels/IRMClassMega.sol

```
# File: src/interestRateModels/IRMClassMega.sol

IRMClassMega.sol:12:              709783723,

IRMClassMega.sol:13:              37689273223,

IRMClassMega.sol:14:              3435973836
```

- src/interestRateModels/IRMClassMidCap.sol

```
# File: src/interestRateModels/IRMClassMidCap.sol

IRMClassMidCap.sol:12:              2767755633,

IRMClassMidCap.sol:13:              40070134595,

IRMClassMidCap.sol:14:              3435973836
```

- src/interestRateModels/IRMClassOHM.sol

```
# File: src/interestRateModels/IRMClassOHM.sol

IRMClassOHM.sol:11:              1546098748700444833,

IRMClassOHM.sol:12:              1231511520,

IRMClassOHM.sol:13:              44415215206,

IRMClassOHM.sol:14:              3435973836
```

- src/interestRateModels/IRMClassStable.sol

```
# File: src/interestRateModels/IRMClassStable.sol
```

```
IRMClassStable.sol:12:              361718388,

IRMClassStable.sol:13:              24123704987,

IRMClassStable.sol:14:              3435973836
```

- src/interestRateModels/IRMClassUSDT.sol

```
# File: src/interestRateModels/IRMClassUSDT.sol

IRMClassUSDT.sol:12:              623991132,

IRMClassUSDT.sol:13:              38032443588,

IRMClassUSDT.sol:14:              3435973836
```

- src/interestRateModels/IRMDefault.sol

```
# File: src/interestRateModels/IRMDefault.sol

IRMDefault.sol:12:              1406417851,

IRMDefault.sol:13:              19050045013,

IRMDefault.sol:14:              2147483648
```

Euler's response: Acknowledged, although stylistically we don't feel it's necessary for the seconds per day constant. The LIDO IRM has been removed, and the other instances in IRMs are auto-generated by a script and anyway aren't easily interpreted by casual readers.

I-08. Duplicate import statements

Affected code:

- src/Synths/ESVault.sol

```
# File: src/Synths/ESVault.sol
```

```
ESVault.sol:11: import {Operations} from "../EVault/shared/types/Types.sol";

ESVault.sol:15: import "../EVault/shared/types/Types.sol";
```

Euler's response: ESVault.sol contract was removed

Gas Optimizations Recommendations

G-01. Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

Consider wrapping with an `unchecked` block where it's certain that there cannot be an underflow

25 gas saved per instance

Affected code:

- src/EVault/modules/Borrowing.sol

```
# File: src/EVault/modules/Borrowing.sol

Borrowing.sol:106:         return collateralBalance - extraCollateralBalance;
```

- src/EVault/modules/Liquidation.sol

```
# File: src/EVault/modules/Liquidation.sol

Liquidation.sol:200:             Assets owedRemaining = liqCache.owed -
liqCache.repay;
```

- src/EVault/modules/Vault.sol

```
# File: src/EVault/modules/Vault.sol

Vault.sol:195:             : balance - marketCache.cash;

Vault.sol:256:                     max = max - used.toShares();

Vault.sol:279:             remainingSupply = marketCache.supplyCap - supply;
```

- src/EVault/shared/BorrowUtils.sol

```
# File: src/EVault/shared/BorrowUtils.sol

BorrowUtils.sol:75:         if ((amount > fromOwed && (amount -
fromOwed).isDust()) ||

BorrowUtils.sol:76:             (amount < fromOwed && (fromOwed -
amount).isDust())) {

BorrowUtils.sol:113:             uint256 change = (owed.toAssetsUp() -
prevOwed.toAssetsUp()).toUint();

BorrowUtils.sol:117:             uint256 change = (prevOwed.toAssetsUp() -
owed.toAssetsUp()).toUint();
```

- src/GenericFactory/GenericFactory.sol

```
# File: src/GenericFactory/GenericFactory.sol

GenericFactory.sol:133:         list = new address[](end - start);

GenericFactory.sol:134:         for (uint256 i; i < end - start; ++i) {
//@audit "end - start" should also be cached in a variable to avoid
re-computation at each iteration
```

- src/interestRateModels/BaseIRMLinearKink.sol

```
# File: src/interestRateModels/BaseIRMLinearKink.sol

BaseIRMLinearKink.sol:27:             ir += slope2 * (utilisation - kink);
```

- src/interestRateModels/IRMClassLido.sol

```
# File: src/interestRateModels/IRMClassLido.sol

IRMClassLido.sol:100:             ir += slope2 * (utilisation - kink);
```

Euler's response: These have been addressed. Some by using a new method `subUnchecked` and others with a plain unchecked block.
Borrowing.sol: The affected code was removed
Liquidation.sol: Fixed here
Vault.sol: Fixed here and here
BorrowUtils.sol: Fixed here and here
GenericFactory.sol: Acknowledged, will keep as is. This code is intended to be called off-chain.
BaseIRMLinearKink.sol, IRMClassLido.sol: Contracts removed

G-02. Use calldata instead of memory for function arguments that do not get mutated

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly bypasses this loop.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gas-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one.

Saves 60 gas per instance

Affected code:

- src/GenericFactory/GenericFactory.sol

```
# File: src/GenericFactory/GenericFactory.sol

- GenericFactory.sol:75:     function createProxy(bool upgradeable, bytes
memory trailingData) external nonReentrant returns (address) {
+ GenericFactory.sol:75:     function createProxy(bool upgradeable, bytes
calldata trailingData) external nonReentrant returns (address) {
```

Euler's response: Acknowledged, will keep as is. Gas saving would only occur during a prohibited reentrancy attempt.

Formal Verification

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

General assumptions / simplifications (Defined in Base.spec):

- The price quote returned by the oracle for all tokens are lower than a large upper bound: 172543658669764094685868896556925636311277724304259663879063105594982 3.
- The lower and upper bound for getQuotes are the same value and equal to the value returned for getQuote
- The quotes returned by getQuote (and getQuotes) are monotonically increasing with the amount.
- The oracle, unitOfAccount, and viewCaller addresses set by governance are nonzero

LTV Configuration assumptions:
These are defined in LTVConfigAssumptions in Base.spec. Note: these assumptions are all about parameters set by the governor who is trusted:

- The liquidation LTV is < 1e4
- The initial Liquidation LTV is < 1e4
- The liquidation liquidationLTV is higher than the borrowLTV
- The initialLiquidationLTV is > liquidationLTV
- The LTV ramp target timestamp less the current block timestamp is greater than the ramp duration

Formal Verification Properties

BalanceForwarder

Module General Assumptions

None other than project-wide assumptions

Module Properties

P-01. enableBalanceForwarder functionality

Status: Verified

Rule Name	Status	Description	Link to rule report
enableBalanceForwarder	Verified	*Calling enableBalanceForwarder will result in balanceForwarderEnabled(account) to be true, where account is the on behalf of account from EVC*	Report

P-02. disableBalanceForwarder functionality

Status: Verified

Rule Name	Status	Description	Link to rule report
disableBalanceForwarder	Verified	*Calling enableBalanceForwarder will result in balanceForwarderEnabled(account) to be false, where account is the on behalf of account from EVC*	Report

RiskManager

Module General Assumptions

None other than project-wide assumptions. Note that The LTVConfig assumptions for the project are particularly relevant to these rules.

Module Properties

P-03. The borrowing LTV is always lower than the liquidation LTV

Status: Verified	Assumptions: • The account has 2 collaterals (so we can bound the loop iterations) • The collateral value for liquidation and borrowing are nonzero

Rule Name	Status	Description	Link to rule report
ltv_borrowing_lower	Verified	*The account liquidity returned for borrowing is always lower than the account liquidity for liquidation*	*Report*

P-04. Account Liquidity Must Revert in Specific Cases

Status: Verified	

Rule Name	Status	Description	Link to rule report
accountLiquidityMustRevert	Verified	*accountLiquidity must revert if either the vault does not control the account or if the oracle is not configured*	*Report*

P-05. Account Liquidity Full must revert in certain cases

Status: Verified

Rule Name	Status	Description	Link to rule report
accountLiquidityFullMustRevert	Verified	*accountLiquidityFull must revert if either the vault does not control the account or if the oracle is not configured*	*Report*

P-06. checkAccountStatus must revert in specific cases

Status: Verified

Rule Name	Status	Description	Link to rule report
checkAccountStatusMustRevert	Verified	*checkAccountStatus must revert unless the caller is the evc and the EVC has configured checksInProgress*	*Report*

P-07. checkVaultStatus must revert in Specific Cases

Status: Verified

Rule Name	Status	Description	Link to rule report
checkVaultStatusMustRevert	Verified	*checVaultStatus must revert unless the caller is the evc and the EVC has configured checksInProgress*	*Report*

Liquidation

Module General Assumptions

None other than project-wide assumptions. Note that The LTVConfig assumptions for the project are particularly relevant to these rules.

Module Properties

P-08. checkLiquidation returns 0 for healthy accounts

Status: Verified	

Rule Name	Status	Description	Link to rule report
checkLiquidation_healthy	Verified	*If checkLiquidation is called for a violator that is healthy (i.e. with accountCollateral greater than accountLiability), it returns 0 for both maxRepay and maxYield*	*Report*

P-09. The borrowing collateral value is lower or equal to the liquidation collateral value

Status: Verified	Assumptions: • The collateral values for borrowing and liquidation are nonzero

Rule Name	Status	Description	Link to rule report
getCollateralValue_borrowing_lower	Verified	*The borrowing collateral value (returned with getCollateralValue with liquidation false) is lower or equal to the liquidation collateral value (returned by getCollateralValue with liquidation true)*	*Report*

P-10. checkLiquidation must revert in certain cases

Status: Verified

Rule Name	Status	Description	Link to rule report
checkLiquidation_mustRevert	Verified	*checkLiquidation must always revert in the following cases:* • *Violator is the same as the liquidator* • *Collateral is not accepted by the vault* • *Collateral is not enabled for the violator* • *The liability vault is not enabled as the only controller of the violator* • *Violator account status check is deferred* • *Price oracle is not configured*	*Report*

P-11. liquidate must revert in certain cases

Status: Verified

Rule Name	Status	Description	Link to rule report
checkLiquidation_mustRevert	Verified	*checkLiquidation must always revert in the following cases:* • *Violator is the same as the liquidator* • *Collateral is not recognized by the vault* • *Collateral is not enabled for the violator* • *The vault does not control the liquidator* • *The vault does not control the violator* • *The status checks are not deferred for the violator* • *The price oracle is not configured*	*Report*

Vault

Module General Assumptions

None other than general project assumptions

Module Properties

P-12. Status Checks are scheduled by the hook operations for the vault

Status: Verified	

Rule Name	Status	Description	Link to rule report
status_checks_scheduled	Verified	*If any of the hook operations (deposit, mint, withdraw, redeem, skim) are called, status checks are scheduled on the EVC*	*Report*

P-13. Balance forwarding is called for all vault operations if it is enabled

Status: Verified	

Rule Name	Status	Description	Link to rule report
status_checks_scheduled	Verified	*If any of the balance forwarding vault functions (deposit, mint, withdraw, redeem, skim) are called with balance forwarding enabled and a nonzero return value, the balance tracker hook is called*	*Report*

Cache

Module General Assumptions

None other than project general assumptions

Identified Bugs

Using the prover and the rule updateVault_no_unexpected_reverts, we identified two bugs which Euler fixed. We were able to verify the rule after Euler's fix. These were the bugs:

- An overflow in the assignment to newTotalBorrows. Counterexample: Report
- An overflow in the conversion of newAccumulatedFees.toShares(). Counterexample: Report

Euler's initial fix was here: commit.
The rule was verified against this fix here: Report (The link below is for a run of the same rule against the most recent commit of the Euler repo at time of writing).
A detailed security analysis of the identified bugs is in this prior section of this report: Section

Module Properties

P-14. Cache updateVault has no unexpected reverts

Status: Formally Verified after Fix	Assumptions: • vaultStorage.lastInterestAccumulatorUpdate < block.timestamp • lastInterestAccumulatorUpdate > 0 • vaultStorage.accumulatedFees < vaultStorage.totalShares

Rule Name	Status	Description	Link to rule report
updateVault_no_unexpected_reverts	Formally verified after fix	*Cache updateVault never reverts under the above assumptions.*	*Report*

ERC4626 Standard Rules

Setup Notes

These rules were proved for a contract that inherits just from the VaultModule and TokenModule simultaneously.

Rule Set General Assumptions

- The following calls do not affect the vault contract's state: invokeHookTarget, requireVaultStatusCheck, tryBalanceTrackerHook, balanceTrackerHook
- RPow.rpow is summarized as a set of axioms defined in GhostPow.spec
- EVC's onbehalfOfAccount returns a fixed nonzero address during any individual call to the vault
- The safeTransferFrom and trySafeTransferFrom functions are summarized as direct calls to a mock ERC20
- The vault itself is not the underlying asset
- For deposit/mint:
    - If the receiver and owner are different addresses the sum of their balances is <= the totalSupply of the vault
    - The balance of the receiver and owner are each separately <= the totalSupply

Rules

P-15. Invariant: the totalAssets are greater than the totalSupply

Status: Verified	

Rule Name	Status	Description	Link to rule report
assetsMoreThanSupply	Verified	*Invariant over all Vault/Token methods that totalAssets >= totalSupply*	*Report*

P-16. For mint/deposit: the contributor's assets must decrease iff the receiver's shares increase

Status: Verified	Assumptions: • The new asset/balance values are leq max_uint256 (this assumption is safe since the calls will revert otherwise) • The vault is not the contributor or receiver account

Rule Name	Status	Description	Link to rule report
contributingProducesShares	Verified	*For the deposit and mint functions, the contributor's assets must decrease if and only if the receiver's shares increase.*	*Report*

P-17. Converting zero shares to assets or zero assets to shares returns zero

Status: Verified	

Rule Name	Status	Description	Link to rule report
conversionOfZero	Verified	*Calling convertToAssets or convertToShares on a zero argument returns 0*	*Report*

P-18. Weak integrity of convertToShares / convertToAssets

Status: Verified	

Rule Name	Status	Description	Link to rule report

conversionWeakIntegrity	Verified	*Converting from shares to assets and then back to shares again (with convertToShares/Assets) results in a value less or equal the original. (They are not necessarily equal due to rounding)*	*Report*

P-19. Weak monotonicity of convertToShares / convertToAssets

Status: Verified

Rule Name	Status	Description	Link to rule report
conversionWeakMonotonicity	Verified	*convertToAssets returns larger values for larger input values. Similar for convertToShares*	*Report*

P-20. Convert to Assets Weak Additivity

Status: Verified

Assumptions:
- The sum of the input share values is less than max_uint128

Rule Name	Status	Description	Link to rule report
convertToAssetsWeakAdditivity	Verified	*converting sharesA and sharesB to assets then summing them must yield a smaller or equal result to summing them then converting*	*Report*

P-21. Convert to Shares Weak Additivity

Status: Verified

Assumptions:
- The sum of the input share values is less than max_uint128

Rule Name	Status	Description	Link to rule report
convertToShares WeakAdditivity	Verified	*converting assetsA and assetsB to shares then summing them must yield a smaller or equal result to summing them then converting*	*Report*

P-22. Monotonicity of shares yielded with respect to assets deposited

Status: Verified	Assumptions: • Excludes the full deposit case (i.e. amount=max_uint256)

Rule Name	Status	Description	Link to rule report
depositMonotonicity	Verified	*when supply tokens outnumber asset tokens, a larger deposit of assets must produce an equal or greater number of shares*	*Report*

P-23. Dust favors the house with underlying asset balance

Status: Verified	

Rule Name	Status	Description	Link to rule report
dustFavorsTheHouse	Verified	*If a user deposits an amount and then immediately redeems the resulting shares, the balance of the vault contract on the underlying asset can only increase as a result.*	*Report*

P-24. Dust favors the house with totalAssets

Status: Verified

Rule Name	Status	Description	Link to rule report
dustFavorsTheHouseAssets	Verified	*If a user deposits an amount and then immediately redeems the resulting shares, the balance of the vault contract on the totalAssets of the vault can only increase*	*Report*

P-25. No supply if no assets

Status: Verified

Rule Name	Status	Description	Link to rule report
noSupplyIfNoAssets	Verified	*If totalAssets is zero, total supply is zero*	*Report*

P-26. Only Contribution Methods Reduce Assets

Status: Verified

Rule Name	Status	Description	Link to rule report
onlyContributionMethodsReduceAssets	Verified	*a user's assets must not go down except on calls to contribution methods*	*Report*

P-27. Reclaiming produces Assets

Status: Verified

Rule Name	Status	Description	Link to rule report
reclaimingProducesAssets	Verified	*an owner's shares must decrease if and only if the receiver's assets increase"*	*Report*

P-28. Redeeming all validity

Status: Verified

Rule Name	Status	Description	Link to rule report
redeemingAllValidity	Verified	*Calling redeem on an address's total balance results in a balance of zero*	*Report*

P-29. The totalSupply is the sum over all balances

Status: Verified

Rule Name	Status	Description	Link to rule report
totalSupplyIsSumOfBalances	Verified	*The total supply is equal to the sum over all user account balances*	*Report*

P-30. Totals Monotonicity

Status: Verified	

Rule Name	Status	Description	Link to rule report
totalsMonotonicity	Verified	*if totalSupply changes by a larger amount, the corresponding change in totalAssets must remain the same or grow* *AND* *equal size changes to totalSupply must yield equal size changes to totalAssets*	*Report*

P-31. Depositing zero results in zero shares

Status: Verified	Assumptions: • Excludes the case where the assets argument to deposit is max_uint256 (deposit all in this implementation). Without this assumption there will be an uninteresting counterexample where the caller has a balance of zero, calls with deposit with the max amount and gets zero shares in return.

Rule Name	Status	Description	Link to rule report
zeroDepositZeroShares	Verified	*Calling deposit will result in zero shares if and only if zero shares are deposited*	*Report*

P-32. The address of the underlying asset never changes

Status: Verified	

Rule Name	Status	Description	Link to rule report
underlyingCannot Change	Verified	*The address of the underlying asset never changes*	*Report*

P-33. The vault is always solvent.

Status: Verified

Rule Name	Status	Description	Link to rule report
vaultSolvency	Verified	*This definition of solvency is split into two parts:* • *We check that the total number of assets exceeds the total number of supply* • *We check that the Vault's internal metric for its assets is less than its actual balance on the underlying asset contract* *For timeout / performance reasons, we needed to verify this for withdraw separately from the other methods and to rewrite it as two rules that handle each of these two parts. Note that first report link passes for all methods aside from withdraw and the second two links are for withdraw.*	*Report for most methods* *Report for withdraw totals clause* *Report for withdraw underlying clause*

"Holy Grail Rule": Accounts Stay Healthy

Setup Notes

This rule shows that if an arbitrary account was healthy initially, if an arbitrary, public non-view function of the vault is called (and does not revert), the account will stay healthy afterwards. Crucially, the account for the health check is an arbitrary one – it may be one related to the function that is called or it may be some other account. As a result this shows that accounts cannot make themselves unhealthy and also that they cannot make other accounts unhealthy

Because the complete EVault contract is large and for verification performance reasons, we verify each of the modules of the Vault separately. For each vault module we verify a contract that inherits from both the module under test and the RiskManager (which is needed to do the health status check). With this setup we cover all methods of the EVault contract.

The table below gives separate reports for each module. We do not include a separate report for the RiskManger module as this is included with all other modules. For the Vault module, we need to run the redeem and withdraw methods separately for performance reasons. For the Liquidation module, we need to run the liquidate method separately and also further decompose liquidate into these cases:

- Case 1: the account of interest is the vault itself
- Case 2: the account of interest is *not* the violator or liquidator
- Case 3: the account of interest is the liquidator but not violator AND:
    - Case 3.1 debt socialization is enabled
    - Case 3.2 debt socialization is disabled

Note that we handle these two cases separately by proving these cases always revert:

- Account == violator (reverts if the "violator" is actually healthy) (see P-08)
- Account == liquidator == violator (reverts on self-liquidation)) (see P-10)

Rule General Assumptions

The project-wide assumptions apply here. In addition, we assume:

- evc.getCollaterals returns the collaterals for a given account in the system and vault.getBalance(user) > 0 ⇔ evc.isCollateralEnabled(vault, account)
- These addresses are distinct: the the collaterals of the health check account, the oracle address, and the unitOfAccount address
- The only way to call the Vault successfully is through the EVC's call/batch
- The following functions all use low-level EVM call ops which the tool cannot automatically route. We use summaries that manually route these to the correct contract:
    - EVC.checkAccountStatusInternal
    - EVC.checkVaultStatusInternal
    - safeTransferFrom
    - enforceCollateralTransfer
    - ProxyUtils.metadata
    - ProxyUtils.useViewCaller
- The following functions are out of scope because they are run once initially or are done by the Governor which we trust: Governance.clearLTV, Governance.setLTV, Initialize.initialize

- Oracle Quotes have a very large upper bound
- We assume the permitted collaterals are ETokens
- The following functions in TrackingRewardStreams.sol do not affect vault state:
    - tryBalanceTrackerHook, balanceTrackerHook
- For any interest rate model used, computeInterestRate does not affect the vault's state

Rules

P-34. accountsStayHealthy_strategy: No function can make a healthy account become unhealthy

Status: Verified	Note: all assumptions are described above

Module / Case Name	Status	Description	Link to rule report
BalanceForwarder	Verified		Report
Borrowing	Verified		Report
Governance	Verified		Report
Initialize	Verified		Report
Token	Verified		Report
Vault – most methods	Verified	*Covers all Vault methods other than withdraw and redeem which are covered in the next 2 cases*	Report
Vault – redeem	Verified		Report
Vault – withdraw	Verified		Report

Liquidation – most methods	Verified	*Covers all Liquidation methods aside from liquidate which is further split into cases and covered by the next 4 table rows.*	*Report*
Liquidation.liquidate Case 1	Verified	*the account of interest is the vault itself*	*Report*
Liquidation.liquidate Case 2	Verified	*the account of interest is not the violator*	*Report*
Liquidation.liquidate Case 3.1	Verified	*the account of interest is the liquidator and debt socialization is enabled*	*Report*
Liquidation.liquidate Case 3.2	Verified	*the account of interest is the liquidator and debt socialization is disabled*	*Report*

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.