OpenZeppelin | security

Euler Vault Kit (EVK) Audit

May 17, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-04-08 To 2024-04-30
Languages	Solidity

Total Issues	32 (16 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	3 (1 resolved)
Low Severity Issues	16 (9 resolved)
Notes & Additional Information	12 (5 resolved, 1 partially resolved)

Scope

We audited the euler-xyz/euler-vault-kit repository at commit 83ea1ad.

In scope were the following files:

```
src
├── EVault
│   ├── DToken.sol
│   ├── Dispatch.sol
│   ├── EVault.sol
│   ├── IEVault.sol
│   ├── modules
│   │   ├── BalanceForwarder.sol
│   │   ├── Borrowing.sol
│   │   ├── Governance.sol
│   │   ├── Initialize.sol
│   │   ├── Liquidation.sol
│   │   ├── RiskManager.sol
│   │   ├── Token.sol
│   │   └── Vault.sol
│   └── shared
│       ├── AssetTransfers.sol
│       ├── BalanceUtils.sol
│       ├── Base.sol
│       ├── BorrowUtils.sol
│       ├── Cache.sol
│       ├── Constants.sol
│       ├── EVCClient.sol
│       ├── Errors.sol
│       ├── Events.sol
│       ├── LTVUtils.sol
│       ├── LiquidityUtils.sol
│       ├── Storage.sol
│       ├── lib
│       │   ├── ConversionHelpers.sol
│       │   ├── ProxyUtils.sol
│       │   ├── RPow.sol
│       │   ├── RevertBytes.sol
│       │   └── SafeERC20Lib.sol
│       └── types
│           ├── AmountCap.sol
│           ├── Assets.sol
│           ├── ConfigAmount.sol
│           ├── Flags.sol
│           ├── LTVConfig.sol
│           ├── LTVType.sol
│           ├── Owed.sol
│           ├── Shares.sol
```

```
│           ├── Snapshot.sol
│           ├── Types.sol
│           ├── UserStorage.sol
│           ├── VaultCache.sol
│           └── VaultStorage.sol
├── GenericFactory
│   ├── BeaconProxy.sol
│   ├── GenericFactory.sol
│   └── MetaProxyDeployer.sol
├── InterestRateModels
│   ├── IIRM.sol
│   └── IRMLinearKink.sol
├── ProtocolConfig
│   ├── IProtocolConfig.sol
│   └── ProtocolConfig.sol
└── interfaces
    ├── IBalanceTracker.sol
    ├── IPermit2.sol
    └── IPriceOracle.sol
```

System Overview

The Euler Vault Kit (EVK) is a framework designed for constructing ERC-4626 vaults with enhanced lending platform functionality. These vaults function as credit vaults, enabling users to earn yield by depositing assets that are subsequently borrowed by borrowers. To borrow, users must deposit a sufficient amount of collateral recognized by the vault, which may be seized in the event of an unhealthy borrower position.

The vault from which the user borrows acts as the controller of the borrower's collateral, continuously increasing the outstanding liability, thus providing yield for depositors. Integration between vaults is facilitated through the Ethereum Vault Connector contract (EVC), which acts as an authentication layer for users and a connector between vaults. The EVC is empowered to move funds on behalf of the users, particularly in the event of liquidation.

Depositing and Withdrawing Collateral

The process of depositing collateral is accomplished through the use of the deposit and mint functionalities of ERC-4626. This entails pulling the underlying asset of the vault from the user and minting a corresponding amount of shares to the user's account.

Withdrawal of collateral is facilitated by the withdraw and redeem functions. In this process, the required amount of shares is burned and the assets are subsequently transferred to the user. In addition, scheduled account and vault status checks are conducted to ensure the healthiness of both the user's position and the state of the vault.

Borrowing

Borrowing is facilitated through vaults that implement controller logic and can act as controllers of the user's collateral. To borrow, the user must possess a sufficient amount of collateral accepted by the vault. Moreover, the user needs to enable the vault as a controller of their collateral through the EVC contract. Borrowing assets is possible up to a value that does not exceed the collateral value adjusted by the configured Loan-to-Value (LTV) ratio. Once the user borrows a certain amount, the debt becomes subject to interest that continually accrues.

Interest Rate Model

Interests are calculated following a linear model with a kink point, similar to the models used by other protocols. Thus, when the utilization rate exceeds a certain threshold, borrowing becomes more expensive.

Liquidations

A position becomes liquidatable once its health score (LTV-weighted liquidity over debt) becomes less than 1. In such a scenario, the smaller the health score, the larger the resulting discount factor (capped at 80%). This factor is used to incentivize liquidators as they will receive collateral assets at such discounted value when repaying the borrower's liability.

It is worth mentioning that during the liquidation process, the violator's position will be forgiven for the account check at the end of the call, as the position might still be under-collateralized. However, it is possible to have positions where, after all the collateral has been used during liquidations, a debt still exists. In such scenarios, the protocol would try to socialize the debt and reduce it from the violator's position.

Governance

The vault also implements governance functionality to enable autonomous administration of its characteristics. An administrator can select which collateral will be recognized by the vault, set margins for the LTV on each asset, determine the interest model to follow, specify the hook target address, set interest fees, and establish supply/borrow caps. All of these factors define the intrinsic properties of the vault and dictate its behavior. In exchange, fees generated from borrows can be split between the protocol and the vault.

Balance Forwarding

The vault can be deployed with the balance tracker contract. If a user enables balance forwarding, the balance tracker contract will receive notifications of any balance changes made by the user. This feature is intended to support the implementation of additional contracts on top of the protocol, which could potentially offer incentives in the form of rewards distributed to the long-term users of the protocol.

Generic Factory

The generic factory enables the creation of vaults through either a beacon proxy or a minimal proxy. The beacon proxy dynamically fetches the implementation address from the factory and then delegates the call to it. This setup allows for the possibility of upgrading the implementation contract directly from the generic factory. On the other hand, vaults created using a minimal proxy do not permit upgrades because the implementation contract is embedded within the code of the proxy contract.

Security Model and Trust Assumptions

The vault may integrate with `Permit2`, which facilitates the transfer of funds from the user to the vault. The `Permit2` contract is trusted to operate in accordance with its specification.

Privileged Roles

Throughout the protocol, there are actors that can achieve sensitive actions that might affect the users' positions:

The `admin` defined in the `ProtocolConfig` can:

- Change the protocol fee receiver
- Change the protocol fee share
- Set the interest fee range
- Set the vault interest fee range
- Set the vault fee config

The `governorAdmin` of the vault can:

- Change the name and the symbol of the vault
- Change the governor admin
- Set the governor fee receiver
- Set the accepted collateral and change its LTV
- Remove accepted collateral
- Set the interest rate model contract

- Set hooks
- Change vault configuration
- Set supply and borrow caps
- Set the interest fee charged to borrowers
- Select the initial Oracle, `unitOfAccount` asset, and underlying asset of the vault

The `upgradeAdmin` defined in `GenericFactory` can:

- Change the implementation contract address which will result in the implementation upgrading for all vaults created using `BeaconProxy`
- Change the `upgradeAdmin` address

Protocol Design and Integration Considerations

The protocol has a few behaviors that might be taken into account when either using it or integrating it with a third-party protocol:

- **Mitigation Actions Can Push Position Into a More Insolvent State:** Depending on the current debt/liquidity status and the vault's configuration values, especially for the LTV, there is a point at which the mechanism to reduce the pressure of the debt and increase the health score can actually get reversed and push the position into a more insolvent state.
- **Liquidator Might Be Allowed to Liquidate the Whole Position:** The current liquidation mechanism offers to repay the debt against discounted collateral once the health score is under 1. However, the implementation allows to liquidate more debt than needed to reach a safe health score once again, with it being possible to liquidate the entire position under the right conditions. This results in the borrower incurring losses greater than necessary to recover their position back to a healthy status.
- **Vault Admin Can Put Positions in an Unhealthy State Immediately:** The Governance allows the admin to call the `clearLTV` function, which would set the LTV to zero and would not recognize such collateral as a valid asset. This results in borrowers backing their debt with such collateral suddenly losing the total collateral value for such an asset, with several of those positions possibly crossing the liquidatable status without any warning. Even though the protocol implements a ramp when lowering the LTV, in this

case, such a ramp does not apply and there is no a grace period to properly smooth the collateralization of the affected accounts.

- **Functions Implement Custom Logic When `type(uint256).max` Is Passed as the Amount:** Several functions use a pattern whereby in case of providing `type(uint256).max` as the amount, the function will use the highest possible value. This might lead to issues when integrating with the vaults.
- **Custom Behavior of `maxWithdraw` and `maxRedeem`:** In the `Vault` contract, the docstring for both `maxWithdraw` and `maxRedeem` states that the functions will "Fetch the maximum amount of assets a user is allowed to withdraw/redeem". However, this is not true as it will call the `maxRedeemInternal` function and after it checks that the user has the collateral enabled and an active controller, it will return zero.
- **The `convertToShares` and `convertToAssets` Might Revert:** The functions `convertToShares` and `convertToAssets` revert in case the amount is big enough that it cannot be encoded within the `Types` library. According to ERC-4626 specification, the `convertToShares` and `convertToAssets` functions must not revert unless there is an integer overflow caused by an unreasonably large input.
- **Dangerous Pattern Used for Flash Loans:** The `flashLoan` function of the `Borrowing` module implements flash loan logic using a dangerous pattern. The initial balance of the asset is saved before transferring the funds to the receiver and executing the `onFlashLoan` function. In the end, it is checked if the current balance of assets is greater or equal to the initial value. This opens the surface for attacks that take advantage of external protocols integrating with the given vault, by executing an action that will increase the balance of the vault. One of those actions could be the vault earning rewards and requiring claiming them from the external contract.
- **Possible Reentrancy in Case the Collateral Allows Executing Code on Transfers:** The pattern used in the liquidation function where the collateral is transferred to the liquidator and the checks are forgiven for the violator leads to a reentrancy possibility. This can happen when the vault whose shares have been seized allows setting up a hook on the receiver during the transfer.
- **Interest Not Being Accumulated due to Overflow:** The calculations made for getting the new interest accumulator might overflow with large values or with long elapses between updates. When this happens, and to prevent the DoS of the vault, the execution continues without updating the interest. This could result in the loss of possible earnings but also the loss of forgiveness of the interest for borrowers.
- **Possible Unexpected Behavior When the Governance Admin Uses the EVC as the Admin:** It is possible for the Governance Admin to misuse the EVC in the Vault. In particular, the Admin could set the EVC as the new Governor Admin to pass through the `governorOnly` modifier during a batch execution, batch operations through the EVC,

and ending with a call in the batch to set back the ownership to the malicious Admin. This might create possible inconsistencies in the accounting system which makes use of the `snapshot`s, the `vaultCache`, and the `vaultStorage` structs while updating a few items on the run.

- **The Rounding in `loop` Function Might Borrow Different Amount of Assets Than Expected:** The `loop` function allows minting shares and the corresponding amount of debt. The passed amount is first converted into shares using `toSharesUp`. To receive the corresponding amount of assets, the shares are converted to assets using `toAssetsUp`. Due to rounding up, it is possible that triggering the `loop` function with the expectation of creating a debt of a given amount may result in an actual debt of a different amount. For instance, when starting with 100 total shares and 101 total assets, and the `loop` call is being made by passing 10 as the amount, the calculations end up increasing the shares by 10 but increasing the borrow by 11. This discrepancy might cause issues during integration if the external contract expects to create a debt of a specific amount whereas the actual debt created is slightly different.
- **Malicious Custom Vault or Module Can Be Used to Initiate External Calls During Execution of Operations:** A custom vault containing malicious code can get adopted and recognized by other Vaults or non-standard modules used by a Vault, such as a custom interest rate model. Afterwards, this vault might be able to initiate external calls once it is called during part of the operation and allow another actor to exploit an attack surface to reenter into the code.

High Severity

H-01 Missing Gap Between Borrow LTV and Liquidation Threshold

The protocol does not implement a gap between the LTV ratio at which borrowers are allowed to borrow and the liquidation threshold at which liquidation can be executed. This means that it is possible to borrow at the very edge of liquidation and be liquidated with a slight price change.

This opens the surface for an attack that could drain the vault of available assets by monitoring price updates. In case the price change is significant enough, the attacker could profit from the change by socializing the debt. When a user becomes eligible for liquidation, a third-party liquidator is rewarded for processing the repayment of their debt with a bonus. This bonus depends on the user's health score, and as the user's health score declines, the bonus grows.

This means that a user's health score is lowered after liquidation if the bonus exceeds the relative over-collateralization of the position. For instance, if a user's position is 10% over-collateralized but the bonus is 11%, then the user's health score declines after liquidation. This scenario allows an attacker, in case of a market stress situation where the price changes significantly, to front run the price change either by monitoring the mempool or through compatible oracles where prices can be updated by anyone like Pyth or Redstone. The attacker could borrow assets, allow the price change, and then liquidate themselves, retrieving all the collateral and profiting by creating bad debt for the protocol. This Proof of Concept was created to illustrate the attack.

Consider introducing a gap between the borrow LTV ratio and the liquidation threshold to prevent borrowers from borrowing assets at an LTV ratio close to liquidation. This gap should be flexible enough to accommodate both highly volatile and barely volatile assets. Specifically, the gap should be wider for high-volatility assets and narrower for low-volatility assets.

Update: *Resolved in pull request #191 of the euler-vault-kit repository and pull request #157 of the ethereum-vault-connector repository. The Euler team stated:*

> *We have made a set of 3 changes to the liquidation system in order to mitigate the issues discovered by our auditors.*

The first issue raised is related to the "Counterproductive Incentives" issue described by OpenZeppelin in their 2019 Compound audit. Liquidation systems that incentivise liquidators with extra collateral value as a bonus (or discount) can, in some circumstances, leave violators more unhealthy than they were pre-liquidation. In the Euler system, the discount is proportional to how unhealthy the user is, which means that in these cases, a liquidator may improve their total yield by performing many small liquidations, rather than one large liquidation. Each smaller liquidation will decrease the user's health and therefore increase their discount for subsequent liquidations, up until the maximum liquidation discount is reached. As described in our Dutch Liquidation Analysis research paper, this scenario can be avoided by selecting an appropriately low maximum discount factor.

Change 1: With this in mind, we have added EVK functionality that allows governors to configure the vault's maximum discount factor. In many cases, governors will compute an appropriate maximum discount based on the highest configured LTV for the vault, although there may be other considerations involved. A governor must specify a value for this parameter, otherwise the liquidation system will not function properly.

The second issue raised is a general observation that price manipulation can be used to attack lending markets, and that some of the oracles we would like to support have special challenges. In particular, pull-based oracles like Pyth and Redstone provide more flexibility to attackers because they can typically choose to use any published prices within an N-minute window. For example, an attacker may be monitoring prices off-chain, waiting for a large decline in the price of a vault's collateral asset (or, equivalently, a large increase in the price of the liability asset). If the decline is sufficiently large, the attacker will search the previous N-minutes of prices and select the pair with the largest difference. The attacker will then submit a transaction that performs the following attack:

- *Updates the oracle with the old price*
- *Deposits collateral and borrows as much as possible*
- *Updates the oracle with the new price, causing the position to become very unhealthy*
- *Liquidates the position from another separate account, leaving bad debt. This bad debt corresponds to profit from the attack at the expense of the vault's depositors*

Although impossible to solve in the general case, to reduce the impact of this issue we have have made two modifications to the EVK:

Change 2: We now allow the governor to configure separate borrowing and liquidation LTVs. This requires the attacker to find correspondingly larger price jumps.

> *Change 3: We have added a "cool-off period" wherein an account cannot be liquidated. Cool-off periods begin once an account has successfully passed an account status check, and last a governor-configurable number of seconds. By setting a non-zero cool-off period, accounts cannot be liquidated inside a block where they were previously healthy.*
>
> *The consequence of this is that the attack described above can no longer be done in an entirely risk-free manner. The position needs to be set up in one block but liquidated in a following block, potentially opening up the opportunity for other unrelated parties to perform the liquidation instead of the attacker. Additionally, such attacks cannot be financed with flash loans. As well as price-oracle related attacks, this protection may also reduce the impact of future unknown protocol attacks.*
>
> *More generally, the cool-off period allows a vault creator to express a minimum-expected liveness period for a particular chain. If the maximum possible censorship time can be estimated, the cool-off period can be configured larger than this, with the trade-off being that legitimate liquidations of new positions may be delayed by this period of time.*

Medium Severity

M-01 Unhealthy Position Locks Collateral That Is Not Recognizable by the Controller

The protocol connects compatible vaults through the Ethereum Vault Connector (EVC) and ensures that sensitive actions triggered by the user on any of the vaults are validated through both account and vault status checks. If the user has a controller enabled, a call is made to ensure that the action does not jeopardize the user's position's health.

However, the issue arises when this check is also conducted in cases where the vault the user interacts with is not recognized as valid collateral for the user's position and has no impact on the user's position. Consequently, this prevents users with unhealthy positions from withdrawing assets or transferring shares of vaults not associated with their positions as the account status check is executed for these operations.

Consider redesigning the logic of managing vaults in a way that allows for the execution of actions on vaults not associated with the user's position. Alternatively, consider thoroughly documenting this behavior.

Update: *Resolved. The Euler team stated:*

> *We acknowledge the issue. When a user enables a vault as a controller for their account, presumably in order to take out a loan, they accept that their access to the account will be limited by the arbitrary rules encoded in the controller.*
>
> *EVaults are implemented to primarily monitor collaterals they are explicitly configured to support, but if the account is unhealthy, a user should accept, as part of the contract with the controller, that any deposit they have in their account may be withheld.*
>
> *Although such deposits do not influence the health score of the position, they may nonetheless be withdrawn by the user in a batch and sold for a recognized collateral, or the liability asset, to bring the account back to health. We have improved the documentation to emphasize this behavior to the users: https://docs.euler.finance/euler-vault-kit-white-paper/#non-collateral-deposits*

M-02 Arbitrary `unitOfAccount` Asset Could Cause Liquidations or Erroneous Value Conversions

The `Liquidation` contract implements the functionality to calculate and liquidate debt positions a user might have. To do so, each vault proxy defines at deployment the `unitOfAccount` asset to be used as the reference. This allows converting two different assets (of supply and borrow) against a third asset to get the value of such positions.

However, the selection of this third asset is arbitrary and meant to be done by the Vault's admin when creating the new vault which, depending on the selection, might have a real impact on the liquidity or the liability of the positions. In particular:

- Assets that might have a popular adoption in the market could get hacked, lose their peg (in the case of stablecoins), or face unexpected behavior that could end up resulting in strong price changes. If these changes ever come closer to zero, the precision could affect the calculations of the protocol, which would translate into wrong conversions between assets and shares, an imprecise health score on accounts, and a higher risk of positions' liquidations (even if in reality those are properly backed with collateral in real value terms).

- The asset used might act as a wrapper for another asset or be a token that has malicious unknown behavior. In such a case, if this asset is used to carry out an underlying attack, or if its intrinsic value drops on purpose to leverage the price pair (e.g., the LP token of a protocol is used, but its behavior can drain the pool, changing its price) then the Vaults attached to that asset will again be subject to the value calculation effect.

Since the `unitOfAccount` asset is not meant to be replaced after the Vault's creation, there is no mitigation for affected vaults to allow positions to migrate unaffected once the `unitOfAccount` asset starts showing the price fluctuations. Moreover, contracts such as the `BaseProductLine` and the `Core` contracts do not perform any checks on the parameters then parsed to the `GenericFactory` contract.

Consider only allowing the usage of trusted and backed-by-industry tokens as the reference token and allowing the protocol to freeze operations and change it if a vault suffers from its effects.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. We agree that the choice of the `unitOfAccount` asset is a crucial security decision, but we think it is a part of the risk management framework and not something that should be enforced in the code. We also agree with the recommendation to only allow trusted tokens as reference assets, but think this filtering should happen at a different level than the vault code, which is un-opinionated and is built on purpose to accept a wide range of configurations. As for making the unit of account configurable, we expect only the most stable assets to pass through the aforementioned filters, in which case the gas costs of reading the configuration from storage, in every operation involving debt, would not be justified.*

M-03 Lack of Incentives to Liquidate Small Positions

There are currently no checks in place to regulate the size of the collateral that users can utilize to open positions. Consequently, users might open positions with collateral that is too small to be profitably liquidated in different vaults. Given the complex architecture and substantial gas costs associated with liquidation, these small positions may never be liquidated, leading to ongoing interest accumulation.

Consider implementing a minimum collateral deposit requirement. This would ensure that the collateral supporting the position maintains sufficient value to incentivize liquidation and overcome the gas cost of such a process.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. We acknowledge that there exists a risk of the vault holding small bad debt positions which are not profitable to liquidate. However, we think that the current code is sufficiently prepared to handle the issue.*
>
> *Firstly, the problem of small positions has been discussed extensively in public, but so far it has not been confirmed to be an issue in practice. Our own experience from V1 confirms this. While we have definitely seen small debt positions which were not picked up by the liquidators, their overall impact on the lending pools was negligible.*
>
> *Secondly, the recommended solution - minimum collateral deposit - can already be mostly enforced with a use of hooks. In the open EVC-EVK architecture, it would make more sense to enforce a minimum limit on the liability rather than collateral deposited, and it could be achieved by hooking operations which manipulate debt in conjunction with* `checkVaultStatus`*.*
>
> *Lastly, with debt socialization, the vault's governor may be incentivized to liquidate the small positions at a loss in order to take the bad debt off the books for the benefit of the vault and its attractiveness to the users.*

Low Severity

L-01 Incorrect Event Emission

The `logBorrowChange` function is expected to emit accurate events based on both the previously owed amount and the currently owed amount. However, within the `increaseBorrow`, `decreaseBorrow`, and `transferBorrow` functions, the `logBorrowChange` function is triggered with the value of `prevOwed` instead of utilizing the currently owed amount retrieved from the `getCurrentOwed` function. This creates a scenario where, if the difference between the amount borrowed or repaid is smaller than the accrued interest, incorrect events may be emitted.

Consider passing the value of the owed amount after considering interest as the `prevOwed` parameter. This will help improve the readability of the logs for both users and off-chain systems.

Update: *Resolved in pull request #160. The Euler team stated:*

> *The fix consists of adding the `InterestAccrued` event which ensures that the `Borrow` and `Repay` events are always emitted in line with the function the user called, and that the amounts of the borrows/repays are included explicitly in the logs.*

L-02 Depositors Can Avoid Participating in Debt Socialization

The `liquidate` function implements logic for debt socialization when there is no more collateral to liquidate. Without delay in exiting positions, lenders can avoid participating in debt socialization by front running liquidation and withdrawing assets.

Consider redesigning the mechanism to ensure that lenders cannot evade participating in debt socialization, either by implementing delayed withdrawals or by incentivizing participation through a reward system that can be implemented using a balance tracker hook.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge this issue. In our view, the largest area of concern is collusion between liquidators and major active depositors: If liquidators were to give/sell advance notice of a liquidation, opportunistic depositors could withdraw beforehand and the remaining passive depositors would take unfairly large haircuts on their deposits. This threat is specifically described in our whitepaper and we have decided to accept the risk.*
>
> *Vault creators who do not wish to accept this risk can enable the `CFG_DONT_SOCIALIZE_DEBT` flag which will mitigate this issue. Although in this case, an alternative strategy of handling bad debt should be designed. Finally, in the future, a custom hook contract could be developed that enforces a withdrawal delay, as suggested.*

L-03 Debt Socialization Can Be Prevented

The current implementation of the liquidation process checks at the end of the liquidation process if the entire debt has been repaid, and if not, whether there is any remaining collateral that can be liquidated. If there is no more collateral to liquidate, the debt might be socialized, depending on the configuration.

However, the issue arises in the `checkNoCollateral` function responsible for checking if there is any collateral left. It iterates over the collateral and checks the balance for each. If the balance is greater than 0, it assumes there is still collateral to liquidate, thereby preventing the

debt from being subject to liquidation. This creates a scenario where any violator who owns just 1 share of the recognized collateral can prevent debt socialization. Consequently, bad debt can continue accruing interest and worsen the situation.

Consider redesigning the debt socialization logic to ensure that the debt is correctly socialized, even in scenarios where just some dust of the collateral is present.

Update: *Resolved. The Euler team stated:*

> *We acknowledge the issue. Since the EVC enforces a limit on the number of collateral assets (and vaults can enforce a lower limit via hooks), somebody who wishes to socialise debt can construct a batch transaction that atomically liquidates each collateral asset without running into significant gas costs: Most of the gas costs will be amortised since storage is warmed by the first liquidation. This method can be used to socialise the debt of an active violator who front-runs 1 wei deposits.*

L-04 Collateral With No Value Can Be Claimed Through Liquidation

The `liquidate` function allows for the claiming of collateral that, according to the configured price oracle, holds no value. Liquidation plays a crucial role in maintaining the stability and solvency of DeFi lending protocols by ensuring that lenders are compensated in the event of borrower default and that the system remains collateralized. However, the liquidator's ability to claim the violator's collateral even when it holds no value does not align with the logic of the liquidation process.

Consider removing the ability to claim the violator's collateral when the price oracle returns zero as its value.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. We agree that removing the collateral without a value without reducing debt does not, on its own, align with the basic liquidation logic, which is to reward the collateral to the liquidator in order to reduce debt.*
>
> *However, when debt socialization is taken into account, removing worthless collateral from the violator's account frees up the bad debt to be removed from the system. In that sense, the behavior does align with the liquidation logic, which is to remove collateral in order to remove debt.*

> *Note that the collateral may be worthless not only because the asset is literally worth nothing according to the oracles, which would be a very unlikely edge-case, but also because the oracle may respond with a zero `amountOut` when the collateral amount is so small that it is not representable in the reference asset. In the latter case, which is much more likely to occur, removing the dust collateral would be the only way to socialize the bad debt.*

L-05 LTV Duality Could Prevent Under-Collateralized Debt Mitigations

Each vault configures the LTV of each acceptable collateral used for taking borrows backed by them. To reduce the pressure when the admin reduces the LTV for a collateral, which could end up in massive liquidations on positions backed by that collateral, the `LTVConfig` library uses a ramp between the old value and the target value. However, if a user has two accounts and wants to temporarily reduce the liquidity pressure while the ramp is still operational, the protocol will not allow it.

Let us consider the following scenario:

- `originalLTV` = `80`
- `targetLTV` = `50`
- `rampDuration` = `1 week`
- `accountA_breakEven` = `75`
- `accountB_breakEven` = `55`

Where the `breakEven` states the point at which the respective account will have a `healthScore` of 1. Originally, both accounts were sufficiently collateralized but after the ramp starts, the ramp's downtrend will eventually liquidate both accounts. The owner of both accounts, with the option to delay the liquidation of the accounts in the search for more collateral to be deposited in the near future (e.g., assets being withdrawn from an L2 bridge whose duration might be similar to 1 week), could prevent the upcoming liquidation of `accountA` by moving debt from `A` to `B` using the `pullDebt` function, which would move both `breakEven` points closer together.

However, because the `requireAccountAndVaultStatusCheck` hook on `accountB` is triggered, the `checkAccountStatus` function from the `RiskManager` contract will be called, passing the execution to the `checkLiquidity` function which uses the `BORROWING` type for getting the LTV of the collateral. This means that all of its valuation will use the `targetLTV` instead of the current value in the ramp, failing the possibility of temporarily

decreasing the pressure of the liquidations, even with the goal of injecting more collateral during the `rampDuration`.

Even though the opposite direction of the debt transfer might not be allowed (compromising accounts with more burden), reducing the pressure on the ones that are closer to the liquidation stage will allow the owner to have more time to find more collateral and have enough for when the ramp ends.

Consider using the LTV ramp when dealing with operations of debt transfer without increasing the actual overall debt to allow users the mitigation of the possible under-collateralized position due to a reduction in the LTV.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. In the current architecture it is not possible to discern what action triggered the account status check, so it would be difficult to perform a special case `pullDebt` during a ramp. While the ramp mechanism goes a long way in preventing user losses in events of LTV decreases, we acknowledge that in the specific edge-case situation described, the user's experience might be sub-optimal and note that instead of 'reducing pressure' the user would be encouraged to close the position.*

L-06 Fees Should Be Distributed Before Change of the Fee Configuration

The `convertFees` function facilitates the distribution of accumulated fees between the protocol and the governor. This distribution is determined by the protocol's fee configuration and the address of the governor receiver. Consequently, any alterations to the fee configuration or the governor receiver address will have a direct impact on the distribution of fees.

Consider triggering `convertFees` function prior to making any changes to the fee distribution configuration or adjusting the governor receiver address to ensure that the accumulated fees are distributed according to the previous configuration.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. The roles of the governor, the fee receiver, and the DAO are not expected to be adversarial towards each other. So, we expect them to not abuse the current fee distribution logic intentionally, but rather cooperate to make sure the interests are aligned.*

> *In the event that any party feels that it may be abused by a change to the settings, they are free to call the `convertFees` function more frequently to lock in the fee distribution more granularly. On the technical side, while a change to governor receiver could be easily handled by the vault, it would be challenging to do the same for the protocol's fee share, which is configured in a different contract.*

L-07 Enum Elements Sorting Is Not Consistent

The `LTVType.sol` file defines the `LTVType` enum. However, the `LTVConfig.sol` file does also define the same `LTVType` enum with the caveat that the order of the elements is inverted. Even though the first file is not used by the protocol, a third-party project might use it as an import, in which case the enum would return the wrong index when queried for the LTV type.

In order to mitigate such integration issues, consider either removing the duplicated file or switching the order of the elements inside the enum to be consistent in all places.

Update: *Resolved in pull request #138.*

L-08 Internal Accounting Mechanism is Incompatible With Tokens That Charge Fees

The vault implementation uses an internal accounting mechanism to keep track of the assets inside the vault and the shares minted during operations such as deposits. However, the same mechanism does not support fee-on-transfer tokens such as Tether (USDT) (the most widely known asset with this feature). This means that in case this asset is used and its feature is turned on, the accounting mechanism will inform more supplied assets than it has, potentially jeopardizing the last users in converting the shares back to the underlying asset.

Consider documenting that these vaults are not meant to be used with such assets, or using the balances before and after the operation to parse the actual values moved into the protocol.

Update: *Resolved. The Euler team stated:*

> *The white paper already documents this fact.*

L-09 Missing Check if the Returned Value From Price Oracle Is Zero

The `GenericFactory` allows the deployment of a vault with any price oracle. This means that there might be vaults created with custom-built oracles that do not correctly check for the calculated value, which could be zero.

Consider validating the value received from price oracle functions such as `getQuote` and `getQuotes` in `getLiabilityValue`, `getCollateralValue`, and `calculateMaxLiquidation` functions to ensure that it is not equal to zero.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. Zero values returned from the oracle calls do not signify an error. In the `IPriceOracle` interface, for the specified `amountIn`, the returned `amountOut` could be zero when either the base token is actually worthless or the value in quote token is not representable, both cases are valid. It is the vault's governor's responsibility to properly select and configure the oracle. The vault itself has no means to verify the oracle's suitability on-chain.*

L-10 Mismatch Between Documentation and Implementation

The NatSpec comment in the `IEVault` interface indicates that the list can contain duplicates. However, in the code, each element in the array is pushed when the `initialized` flag from the `LTVConfig` struct is down, and there is no mechanism provided to reset it back to `false`, even after calling the `clearLTV` function.

To avoid potential integration issues for other protocols, it is advisable to update the documentation to accurately reflect the current behavior.

Update: *Resolved in pull request #139.*

L-11 Code-in-Address Does Not Guarantee Compatibility

In a few cases, the protocol checks if an address possesses code as a validation before proceeding with the assignment. However, this does not guarantee that it could be another contract implementing a totally different logic. In particular:

- The `initialize` function from the `Initialize` contract module has code in what is supposed to be the Vault's underlying asset. However, it does not necessarily mean it is a compatible ERC-20 token.
- The `setHookConfig` function does not verify if the `newHookTarget` implements the expected methods and complies with the rest of the hooks.

Consider using an introspection standard on the protocol-complementary contracts that interact with the Vaults, and take into consideration that addresses with code added as assets might not have a compliant ERC-20 token behind them.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. We acknowledge that checking for the existence of code at an address does not guarantee that the contract conforms to the required interface. The checks are not intended to provide guarantees, they are performed as a basic validation, making sure the user did not make an obvious mistake.*
>
> *The check is a small improvement over a common practice to check for the zero address. We do not see a need to use introspection standards, which themselves do not provide strong guarantees about a contract's interface or implemented behavior. If the complementary contract does not implement the required functions, the vault will simply not function.*

L-12 Unhandled Call Output

The `BalanceUtils` contract implements methods to handle share balances and allowances. When balances are changed, the `tryBalanceTrackerHook` function is called which makes an external call to the `BalanceTracker` contract.

However, the function returns an unhandled `success` output which is also not being handled in the methods that use it. Even though the name implies that it might be possible to have a non-successful call and continue with the execution, there is no explicit handling mechanism to catch the situation and inform the user or the protocol.

Consider handling such output in each function that makes use of the `tryBalanceTrackerHook` function, at least for logging the failed hook call so users can be aware of it.

Update: *Resolved in pull request #140. The Euler team stated:*

> *We have decided to not attempt to handle failed calls to the balance tracker contract. Instead, we will be using Certora's formal verification tools to prove that the balance tracker implementation we will be using will not revert.*

L-13 Missing Docstrings

Throughout the codebase, there are multiple code instances that do not have docstrings:

- The entire `BeaconProxy` contract in `BeaconProxy.sol`
- The `onFlashLoan` function in `Borrowing.sol`
- The entire `DToken` contract in `DToken.sol`
- The entire `Dispatch` contract in `Dispatch.sol`
- The entire `EVault` contract in `EVault.sol`
- The `IComponent` interface and the entire `GenericFactory` contract in `GenericFactory.sol`
- The events in `Governance` contract in `Governance.sol`
- The `IIRM` interface in `IIRM.sol`
- The entire `IPermit2` interface in `IPermit2.sol`
- The entire `IPriceOracle` interface in `IPriceOracle.sol`
- The `IProtocolConfig` interface in `IProtocolConfig.sol`
- The entire `IRMLinearKink` contract in `IRMLinearKink.sol`
- The events defined in `ProtocolConfig` contract in `ProtocolConfig.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #149.*

L-14 Incomplete Docstrings

Throughout the codebase, there are several instances of incomplete docstrings:

- The interfaces defined in `IEVault.sol` are utilizing docstrings. However, not all functions are completely documented, lacking documentation for parameters and return values.
- The function declared in the `IProtocolConfig` interface does not document all parameters of the functions.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #152.*

L-15 Missing Input Validation

Throughout the codebase, there are multiple instances of missing input validation:

- Missing zero address check for the `admin_` and `feeReceiver_` parameters in `ProtocolConfig`'s contract constructor.
- Missing zero address check for the `admin` parameter in `GenericFactory`'s contract constructor.

Consider implementing input validation for the listed instances to prevent unexpected behavior.

Update: *Resolved in pull request #142.*

L-16 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled. Throughout the codebase, there are multiple floating pragma directives of `solidity ^0.8.0`.

Consider using fixed pragma directives.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. We consider the Solidity pragma to specify the language version that the code was written for and its purpose is not to enforce a minimum*

> *compiler. As the contracts are meant to be a reusable kit, we prefer to leave the pragma at the lowest version that we expect this code to compile against. The compiler version will be chosen and locked in when the contracts are deployed, and we of course recommend to use the latest compiler version available at the time.*

Notes & Additional Information

N-01 Missing Named Parameters in Mappings

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of a mapping's purpose.

Throughout the codebase, there are multiple mappings without named parameters:

- The `proxyLookup` state variable in the `GenericFactory` contract
- The `_interestFeeRanges` state variable in the `ProtocolConfig` contract
- The `_protocolFeeConfig` state variable in the `ProtocolConfig` contract

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. In the listed instances, the mapping targets are custom structs, whose names sufficiently specify the type of data the mapping holds. Naming the parameters would be redundant in these cases.*

N-02 Typographical Errors

Consider addressing the following typographical errors:

In `ProtocolConfig.sol`:

- The parameter in the `SetFeeConfigSetting` event should be `vault` not `ault`.

- In the NatSpec comment, it should be `interest` not `intereset`.

In `Events.sol`:

- In the NatSpec comment for the `repayAssets` param, it should be `transferred` instead of `transfered`.
- In the NatSpec comment for the `yieldBalance` param, it should be `transferred` instead of `transfered` .

In `Vault.sol`:

- In the comment, it should be `withheld` not `witheld`.

In `Cache.sol`:

- In the comment, there is confusing information about `MarkeStorage`.

Update: *Resolved in pull request #143.*

N-03 Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease code clarity and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity file or when inheritance chains are long.

Throughout the codebase, global imports are being used:

- The series of imports in `Types.sol`
- The import of `Types` in `AssetTransfers.sol`, `BalanceUtils.sol`, `Base.sol`, `BorrowUtils.sol`, `Borrowing.sol`, `Cache.sol`, `Governance.sol`, `Initialize.sol`, `LTVUtils.sol`, `Liquidation.sol`, `LiquidityUtils`, `RiskManager.sol`, `Token.sol`, and `Vault.sol`
- The import of `Constants` in `Assets.sol`, `ConfigAmount.sol`, `Dispatch.sol`, `EVCClient.sol`, `Initialize.sol`, `Owed.sol`, and `ProxyUtils.sol`
- The import of `IIRM` in `IRMLinearKink.sol`
- The import of `IProtocolConfig` in `ProtocolConfig.sol`
- The import of `Errors` in `RevertBytes.sol`

Following the principle that clearer code is better code, consider using the named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. While we agree that named imports are generally advised, we believe that in certain cases, importing all of the contents of the file is more readable and concise. We believe that for the listed instances.*

N-04 Unnecessary Cast

Within the `EVCClient` contract, the `IEVC(evc)` cast is unnecessary.

To improve the overall clarity, intent, and readability of the codebase, consider removing unnecessary casts.

Update: *Resolved in pull request #144.*

N-05 Use Custom Error Parameters

Throughout the codebase, custom errors have been used to inform where an assertion or condition has not been met. However, none of the implemented custom errors use parameters. Being able to pass critical information to these parameters (such as addresses, values, or other criteria) provides a significant advantage when debugging and examining the details of a reverted call.

To add more detail to the failing transactions, consider using parameters alongside the custom errors.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. In our opinion, in the majority of errors in the codebase, adding parameters would not bring much benefit and we opted for a simple error pattern. Note that there is one exception: in `SafeERC20Lib`, the `E_TransferFromFailed` includes errors from both transfer attempts, which could be useful for UIs.*

N-06 Unsafe ABI Encoding

It is not an uncommon practice to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate calldata for a low-level call. However, the first

option is not typo-safe and the second option is not type-safe. The result is that both of these methods are error-prone and should be considered unsafe.

Within `SafeERC20Lib.sol`, there are multiples uses of unsafe ABI encodings:

- The use of `abi.encodeWithSelector` in `trySafeTransferFrom` function
- The use of `abi.encodeWithSelector` in `safeTransfer` function

Consider replacing all the occurrences of unsafe ABI encodings with `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos.

Update: *Resolved in pull request #145.*

N-07 State Variable Visibility Not Explicitly Declared

Throughout the codebase, there are state variables that lack an explicitly declared visibility:

- The `protocolConfig`, `balanceTracker`, and `permit2` variables in `Base.sol`
- The `BEACON_SLOT`, `IMPLEMENTATION_SELECTOR`, `MAX_TRAILING_DATA_LENGTH`, `beacon`, `metadataLength`, `metadata0`, `metadata1`, `metadata2`, and `metadata3` variables in `BeaconProxy.sol`
- The `VIRTUAL_DEPOSIT_AMOUNT` variable in `ConversionHelpers.sol`
- The `evc` variable in `EVCClient.sol`
- The `REENTRANCYLOCK__UNLOCKED`, `REENTRANCYLOCK__LOCKED`, `MAX_PROTOCOL_FEE_SHARE`, `GUARANTEED_INTEREST_FEE_MIN`, and `GUARANTEED_INTEREST_FEE_MAX` variables in `Governance.sol`
- The `INITIAL_INTEREST_ACCUMULATOR` and `DEFAULT_INTEREST_FEE` variables in `Initialize.sol`
- The `MAXIMUM_LIQUIDATION_DISCOUNT` variable in `Liquidation.sol`

For clarity, consider always explicitly declaring the visibility of variables, even when the default visibility matches the intended visibility.

Update: *Resolved in pull request #146.*

N-08 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are no contracts that specify a security contact.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #147.*

N-09 Magic Numbers

Throughout the codebase, there are instances where explicit values are used directly in arithmetic operations:

- `1e4` in the `ConfigAmountLib` library
- `1e4` in the `Governance` contract
- `1e18` in the `Liquidation` contract
- `1e27` in the `Cache` contract

In order to improve the readability of the codebase, consider using a constant to define such values and document their purpose.

Update: *Partially resolved in pull request #148. The Euler team stated:*

> *Fixed for `1e4`, acknowledged for `1e18` and `1e27`.*

N-10 Overflow in `setOwed` of `UserStorage` Library

The `UserStorage` library tries to fit a `uint144`-sized value into a field of size `2**143`, leading to an overflow. While the protocol handles the necessary checks outside the library, the library itself remains vulnerable to overflow.

Consider incorporating essential checks within the `UserStorage` library to ensure its safe standalone usage.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. The `UserStorageLib` is coupled with the `Owed` type, not with its underlying `uint144`. One of the purposes of the `Owed` type is to make sure that the custom data size is enforced (`2 ** 143`).*

N-11 Liquidators' Yield Might Be Zero

In the `Liquidation` contract module, the calculation done for getting the yield when repaying a certain amount might round the yield received by the liquidator down to zero. This can happen when the `desiredRepay` value is rather small compared to the `maxRepay` value and the conversion between both assets is not enough.

Even though there is a check against the `minYieldBalance` value to receive, the liquidator might default such input to zero. As such, the liquidator would end up repaying some debt and receiving nothing.

Consider rounding up the yield to be repaid to the liquidator during such an operation. This will help prevent such a scenario from happening in edge conditions and would incentivize liquidations.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. For the sake of simplicity, we prefer to allow this edge case. As mentioned, `minYieldBalance` should be used by the liquidator. If they forget to set it, the proposed solution would provide them with a single unit of collateral, which most probably would not change much for the profitability of the liquidation.*

N-12 Missing Functionality for Recovering Tokens

The Vault lacks functionality for recovering tokens that were accidentally sent to it. Since one way to deposit assets is to transfer tokens to the Vault and then call the `skim` function, it is expected that users might mistakenly transfer certain tokens to the wrong vault and then be unable to recover those tokens.

Consider implementing a `sweepTokens` function that would allow the recovery of tokens that are not the underlying asset for that vault.

Update: *Acknowledged, not resolved. The Euler team stated:*

> *We acknowledge the issue. The fact that the `skim` function exists does not mean that the users are encouraged to simply send the tokens into the vault. Even if they send the correct asset to the vault, but do not claim them immediately, they risk the assets being taken over by someone else, presumably a bot looking for mistakes like this. So, if they do make a mistake in the vault address, but otherwise follow the intended logic, the transaction would not be successful.*
>
> *Sweep functions have been found to carry security risks of their own. One way other systems approach this issue is to make the sweep function only callable by an admin, but we expect most EVaults to be non-governed. If the sweep function were permissionless, then just a single bot looking for opportunities in all the existing EVaults could undermine its intended use.*

Conclusion

The Euler Vault Kit (EVK) is a structured framework tailored to build ERC-4626 vaults integrated with advanced lending platform features. These vaults serve as credit facilities, allowing users to generate yield by depositing assets that are then borrowed by other users.

This audit was conducted over the course of three weeks and revealed one high- and three medium-severity issues. Various recommendations have been provided to enhance the quality and documentation of the codebase. The well-implemented test suite, along with the fuzzing setup, showed that the codebase is very mature. The Euler team was very supportive in answering questions in a timely manner.