PRIVATE

Code Assessment

of the Euler Vault Kit Smart Contracts

18 June, 2024

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Euler with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Euler Vault Kit according to Scope to support you in forming an opinion on their security risks.

Euler implements Euler Vault Kit, a system for building lending vaults where lenders can earn interest on their deposited assets and borrowers can borrow the deposited assets against collateral. The system is designed to be modular, allowing the creation of lending markets with flexible configurations.

The most critical subjects covered in our audit are liquidation rewards, functional correctness and precision of arithmetic operations.

Security regarding liquidation incentives is high, an unexpected peculiarity was identified where the system can incentivize liquidators to perform multiple partial liquidations instead of a single full liquidation (see Multiple partial Liquidations can result in higher than expected discount). Security regarding functional correctness and arithmetic precision are also high.

The general subjects covered are liveness, solvency, and access control. Security regarding all the aforementioned subjects is high.

During the review by ChainSecurity, issues identified by other concurrent auditors were disclosed before they could be found by ChainSecurity. Those issues are not included in the report, and we are unable to tell whether they would have been found or not.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	5
• Risk Accepted	4
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Euler Vault Kit repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	22 April 2024	ae7795f7fda71c18bb79805b29bb9fd4634a5930	Initial Version
2	13 June 2024	55d1a1fd7d572372f1c8b9f58aba0604bda3ca4f	first round of fixes

For the solidity smart contracts, the compiler version `0.8.23` was chosen.

2.1.1 Included in scope

This report covers the Euler EVault base contracts, the factory and Synth Vaults.

- src/EVault/Dispatch.sol
- src/EVault/DToken.sol
- src/EVault/EVault.sol
- src/EVault/IEVault.sol
- src/EVault/modules/BalanceForwarder.sol
- src/EVault/modules/Borrowing.sol
- src/EVault/modules/Governance.sol
- src/EVault/modules/Initialize.sol
- src/EVault/modules/Liquidation.sol
- src/EVault/modules/RiskManager.sol
- src/EVault/modules/Token.sol
- src/EVault/modules/Vault.sol
- src/EVault/shared/AssetTransfers.sol
- src/EVault/shared/BalanceUtils.sol
- src/EVault/shared/Base.sol
- src/EVault/shared/BorrowUtils.sol
- src/EVault/shared/Cache.sol
- src/EVault/shared/Constants.sol
- src/EVault/shared/Errors.sol
- src/EVault/shared/EVCClient.sol
- src/EVault/shared/Events.sol

- src/EVault/shared/LiquidityUtils.sol
- src/EVault/shared/LTVUtils.sol
- src/EVault/shared/Storage.sol
- src/EVault/shared/lib/AddressUtils.sol
- src/EVault/shared/lib/ConversionHelpers.sol
- src/EVault/shared/lib/ProxyUtils.sol
- src/EVault/shared/lib/RevertBytes.sol
- src/EVault/shared/lib/RPow.sol
- src/EVault/shared/lib/SafeERC20Lib.sol
- src/EVault/shared/types/AmountCap.sol
- src/EVault/shared/types/Assets.sol
- src/EVault/shared/types/ConfigAmount.sol
- src/EVault/shared/types/Flags.sol
- src/EVault/shared/types/LTVConfig.sol
- src/EVault/shared/types/Owed.sol
- src/EVault/shared/types/Shares.sol
- src/EVault/shared/types/Snapshot.sol
- src/EVault/shared/types/Types.sol
- src/EVault/shared/types/UserStorage.sol
- src/EVault/shared/types/VaultCache.sol
- src/EVault/shared/types/VaultStorage.sol
- src/InterestRateModels/IIRM.sol
- src/InterestRateModels/IRMLinearKink.sol
- src/interfaces/IBalanceTracker.sol
- src/interfaces/IFlashLoan.sol
- src/interfaces/IHookTarget.sol
- src/interfaces/IPermit2.sol
- src/interfaces/IPriceOracle.sol
- src/ProductLines/BaseProductLine.sol
- src/ProductLines/Core.sol
- src/ProductLines/Escrow.sol
- src/ProtocolConfig/IProtocolConfig.sol
- src/ProtocolConfig/ProtocolConfig.sol
- src/GenericFactory/BeaconProxy.sol
- src/GenericFactory/GenericFactory.sol
- src/GenericFactory/MetaProxyDeployer.sol
- src/Synths/ERC20Collateral.sol
- src/Synths/ESynth.sol
- src/Synths/EulerSavingsRate.sol

- src/Synths/IRMSynth.sol
- src/Synths/PegStabilityModule.sol

2.1.2 Excluded from scope

Any contracts inside the repository that are not mentioned in `Scope` are not part of this assessment. Third party dependencies are assumed to be correct and behave according to their specification.

Tests and deployment scripts are excluded from the scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Euler offers Euler Vault Kit, a set of smart contracts that leverage the Euler Vault Connector to implement flexible and customizable lending markets.

2.2.1 Liquidity provision

Liquidity providers can deposit the underlying asset into Euler vaults in exchange for shares of the vault. The goal for LPs is to earn interest on their deposit, which comes from the interest paid by borrowers that borrow the underlying asset. The balance of the underlying asset is either in cash, which earns no interest or is lent out. The total assets owned by the vault follow the equation `totalAssets = cash + totalBorrows`. The `totalBorrows` part of the equation accrues interest according to the Interest Rate Model configured in the vault, therefore the `totalAssets` increase accordingly. Since each share owned by the Liquidity Providers is backed partly by borrows, which earn interest, and partly by cash, which sits idly in the contract, the interest earned on deposits is lower than the interest paid by borrowers, and follows the relation: `supplyRate = borrowRate * totalBorrows / totalAssets`.

To prevent donations attacks, and share price manipulation, the vault contract prevents donations, so that the `cash` reserve can only increase through deposits and not direct transfers. Any excess balance in the contract can be deposited through the `skim()` method, granting shares to the caller. The initial share price is set to 1:1 with the underlying asset because a virtual balance of 10^6 assets and 10^6 shares are added to the total balances during vault operations. The virtual shares earn a fraction of the interest, which is therefore lost. In general, the virtual shares are many order of magnitudes fewer than real shares, since only 10^6 wei of virtual shares are emitted, therefore the loss should be negligible unless the underlying token has high value per wei (e.g. GUSD which has 2 decimals).

The operations that are relevant to liquidity provision are:

- `deposit()`
  deposits a given amount of underlying and mints the corresponding shares.
- `mint()`
  mints a given amount of shares and deposits the corresponding underlying.
- `withdraw()`
  withdraws a given amount of underlying, and burns the corresponding shares.
- `redeem()`
  burns a given amount of shares, and withdraws the corresponding underlying.

2.2.2 Borrowing and interest rates

On the opposite side to liquidity provision, users can borrow the liquidity that is available in the vault, provided that they have enough collateral. To keep track of the interest accrued by the vault, a global interest accumulator is updated when the first operation in a block is performed. The interest accumulator is initialized to 10^{27} when the vault is created. On each vault operation, the `initOperation()` internal method is called, which is also responsible for accruing the interest rate. If any amount of time has elapsed since the last time the interest has accrued, the stored interest rate is used to increase the stored interest accumulator, using the following rule:

$$I_{n+1} = I_n(1 + r_n)^{t_{n+1} - t_n}$$

That is: the new accumulator value (I_{n+1}) is equal to the old one times the per-second rate to the difference in seconds between the current time and the last update. The position of every individual borrower is represented by the global accumulator value at the time of loan creation, and the initial owed amount. Since every user shares the same interest rate model, the global accumulator allows to calculate the owed amount of a user at any time as `owed_t = interestAccumulator_t / interestAccumulator_0 * owed_0`. Amounts of debt are internally represented in extended precision, where `1` wei of debt in the underlying asset is represented as 2^{31}. This extended precision allows to correctly accrue interest even on small positions, or tokens with low decimals, without incurring significant rounding issues.

The methods that interact with borrowing are:

- `borrow()`

  `borrow()` is the main borrowing interface, it transfers the underlying asset to the caller/recipient, and increases the amount owed by the caller.

- `loop()`

  `loop()` mints shares of the vault by increasing the caller's owed amount. It is equivalent to performing a borrow and then depositing the shares.

- `pullDebt()`

  transfers the debt of a user to the caller.

- `repay()`

  it transfers the underlying asset back to the vault, and clears the corresponding amount of debt for the caller/recipient.

- `deloop()`

  repays a debt amount with vault shares.

- `flashLoan()`

  performs a flash loan. It does not interact with the accounting internals of the vault, and to minimize gas consumption it does not perform the global interest accrual nor the vault and account checks. It also does not perform the call through the EVC.

Methods that increase the debt amounts must enforce the collateralization of the loans. For this reason, an account status check is requested to be performed once an operation or a batch of operations are done. The account status check is scheduled in the Euler Vault Connector, from which calls vaults are originating. After a batch of operations in EVC, a call to the vault's method `checkAccountStatus()` ensures the collateralization of accounts whose debt had increased.

2.2.3 Collateralization

As in most decentralized lending protocols, Euler Vault Kit assures the solvency of loans by their over-collateralization. Every amount of debt issued by vaults is backed by a greater (or equal) value of collateral, provided in the form of possibly many collateral tokens enabled by Governance. Each enabled collateral currency has a given *Loan To Value* (LTV), which is a number in the `[0, 1]` range that parametrizes how much debt can be issued for a unit of the given collateral. The health of a position is the ratio between the collateral liquidity value, that is the value of each collateral weighted by its LTV, and the value of the debt:

$$h = \frac{\sum_{i=1}^{k} c_i v_i}{d}$$

Where h is the health, c_i and v_i are respectively the value of the i-th collateral and its LTV, and d is the total debt.

When the health declines below 1, because interest accrues, or because of market movements the value of the collaterals decrease with respect to the value of the debt, the position can be liquidated. A liquidation allows a third-party liquidator to repay the loan, and acquire an amount of collateral of higher value than the repay. This discount applied on collateral acquired during liquidations incentivizes arbitrageurs to act as liquidators and keep the loans in the system overcollateralized at all time.

Collaterals used in a vault are not deposited into the vault, but stay in the possession the the borrower. For this reason, they have to be compatible with the EVK. This means that, in the smart contract implementing the collateral token, every method that reduces the collateral balance of a user needs to request an account status check for the user in the Euler Vault Connector. The vault from which the user is borrowing is registered in the EVC as controller, and the EVC can therefore perform the solvency check every time a collateral balance changes.

2.2.4 Pricing of Liability and Collaterals

Euler Vault Kit requires specifying a *unit of account* at the time of vault creation, which is the reference asset that will be used to price the debt (liability) and the collaterals. The pricing is done through Euler Price Oracles, an adapter interface that allows Euler Vault Kit to interact with several kinds of oracles. Each vault is configured with one oracle address, that will be queried for the pricing of the debt and collaterals. The price oracle exposes two methods: `getQuote(uint256 amount, address base, address quote)` and `getQuotes(uint256 amount, address base, address quote)`. Arguments of the call are the `amount` of base asset that needs to be priced, the `base` asset address, and the `quote` asset address, that is the asset in which the price is expressed.

The difference between `getQuote()` and `getQuotes()` is in the number of outputs, and its interpretation. `getQuotes()` returns two values, that correspond to the value of `amount` according to the bid and ask prices. These prices are meant to offer a lower and an upper bound on the prices that can be realized on the markets. The bid price is the price that we can expect to realize when selling `amount` of `base` asset, therefore the price will be lower because of the bid-ask spread, and of the slippage that will be incurred when selling. Conversely, the ask price is the price that can be expected when buying `amount` of `base` token and will be higher. The second method `getQuote()` returns a single value, which is to be considered the *mid-point* price. The mid-point price is between the bid price and the ask price, and is used when pricing liability and collaterals to compute the health factor for liquidations.

During loan creation, the `bid` price is used to price collaterals, and the `ask` price is used to price the debt. The rationale is that the collaterals can be sold to cover the debt, so when sold they can obtain the *bid* price, which is the lower of the price range. Similarly, the debt has to be acquired on the market to be repaid, so the *ask* price is used, because it is the price to buy the amount of debt needed to repay the loan.

The *bid-ask spread* provides pessimistic estimates on prices for loan creation, such that in times of market volatility, or if the liquidity of a market is low, a conservative amount of debt can be loaned, which decreases the chances of generating bad debt through loan creation.

2.2.5 Liquidations

In decentralized lending protocols, liquidators are incentivized to repay unhealthy positions in exchange for part of the borrower's collateral. In general, in exchange for the repayment of a given amount of debt, a greater value of collateral is given. This incentivizes arbitrageurs to perform liquidations since a risk-free profit can be realized. The incentive scheme implemented by Euler Vault Kit is in the form of a Dutch auction-based discount offered on the collateral of the unhealthy position. When the *health* of the position is exactly just below 1, the discount offered is 0%. As the position's health decreases, the discount will increase following the relation:

$$D = 1 - h$$

so that, for example, if the health of the position is 0.9, a 10% discount is offered on the position's collateral to liquidators. When the health of a position gradually decreases, because of collateral prices falling with respect to liability prices, an increasing discount will be offered, which incentivizes arbitrageurs in a gradually increasing manner. The liquidation will then be performed as soon as arbitrageurs deem it profitable, as the game theoretical equilibrium for them is to take profit, however small it might be, since otherwise they will lose it to some other arbitrageur. This incentive scheme should be flexible enough to provide big discounts that are necessary in times of market illiquidity/volatility, while on average performing liquidations at low discounts which imply a smaller loss for the position holders.

2.2.6 LTV Governance

The LTV of a collateral is configurable by Governance. A new collateral is initialized by setting its LTV. The LTV of an existing collateral can be changed by Governance. If the LTV is increased, the change is immediate. However, if the LTV is decreased, Governance can choose to gradually decrease the LTV from the initial value to the new value ("ramping"). This allows borrowers whose collateral's LTV is being reduced to increase their collateral or repay the debt, to prevent being liquidated. The LTV of problematic or malicious collaterals can be *cleared*, which is different than setting the LTV to zero. Setting the LTV to zero still recognizes the collateral as part of the vault, and that collateral can reclaimed during a liquidation, and bad debt is not socialized while the violator still owns this collateral, even if its LTV is set to zero. When governance *clears* a collateral, the collateral stops participating in the health calculation, it cannot be liquidated during liquidations, and the violator still having a positive balance of cleared collateral does not prevent bad debt socialization.

2.2.7 Interest rate models

The interest rate applied to vault loans is a variable interest that depends on the utilization, the ratio between debt and total assets of the vault. *IRMLinearKink* implements the default interest rate model of vaults, and is configured at creation time by a `baseRate`, the rate that is applied on borrows at 0% utilization, `slope1` coefficient, the slope of the linear increase of the interest rate with utilization up to where utilization reaches the `kink` value. At the `kink` value, `slope2` drives the increase in interest rate. `slope2` is higher than `slope1`, and incentivizes new suppliers to deposit to decrease the utilization, or borrowers to repay their debt, after `kink` utilization is reached.

2.2.8 Balance Forwarder

Shareholders of *EVault* can enable their balance to be forwarded to the *BalanceTracker* contract, to participate in liquidity mining incentives. Every operation that modifies the share balance of the user will forward the new balance to the *BalanceTracker*. Users can decide not to forward their balance in order to spend less gas on such operations. During liquidations, the `controlCollateralInProgress` flag of the Euler Vault Connector is set and is passed as an argument to the *BalanceTracker*. The balance tracker can choose to forfeit the rewards when the flag is set so that the gas cost of transfers in liquidations is not increased by potential rewards.

2.2.9 Fees

Interest earned on borrows is in part earned by the fee receivers, who are the vault fee receivers, set by the vault Governance and the protocol fee receiver, who represents Euler. The percentage of the fee that goes to fee receivers is set by vault Governance, and if it is outside of the range from 10% to 100%, it is validated by the ProtocolConfig contract, a contract controlled by Euler governance. On every interest accrual, the fee portion of the interest is accumulated into the `accumulatedFees` storage variable in the form of shares that slightly dilute the other shareholders. The `totalShares` storage variable is also incremented, but not the balances of the receivers yet. The fee is finally transferred to the balance of the receivers by calling the `convertFees()` method in the *Governance* module.

2.2.10 Operations internals

All methods that change the balance of debt, underlying, or shares of a user are referred to as operations. Operations are always called with the `callThroughEVC` modifier, which creates an EVC `call()` for the operation where checks are deferred at the end, if it is not already the case, such that the vault and account checks scheduled in `initOperation()` are always performed after the operation.

Every vault operation (transfers, borrows, deposits, liquidations) calls `initOperation()` as a first step. `initOperation()` is responsible for the following:

- **Authentication**

  In Euler Vault Kit, the account that performs the operation is not necessarily `msg.sender`. Authentication is performed by the EVC, which sets the account on behalf of which the call is performed. `initOperation()` therefore queries the EVC to know which account is performing the call.

- **Controller check**

  Increasing the debt of an account requires the vault to be enabled as the controller of the account. The *controller* is a privileged address that can use the EVC to act on behalf of the controlled account. In the case of vaults, this is required to transfer collateral during liquidations. If the caller has not enabled the vault as its controller, the vault can't issue debt to the caller. `initOperation()` therefore queries the EVC to check if the caller has enabled the vault as controller when the operation performed is one that increases the liability.

- **Vault upkeep**

  The total debt of the vault accrues interest, and the newly computed values are written to storage. If supply and borrow caps are set, a snapshot of the current vault state is saved in storage. Its purpose is to compare this snapshot with the state after the operation has been performed to detect if the supply or borrow caps have been exceeded because of the operation.

- **Hook call**

  Operations can be configured to call a global `hookTarget` contract, to which the operation payload is forwarded. The *hook* contract has the power to block the operation, by reverting the call, for example, if certain conditions are not met.

- **Solvency check**

  Some operations will increase the liability of the caller, such as borrows and liquidations, or possibly decrease the collateral balance, such as transfers or withdrawals. Following these operations, a solvency check has to be performed on the account that is concerned, as they might not be solvent anymore. In the call to `initOperation()`, the account for which the solvency check is necessary can be specified, by passing either an address or the special constant `CHECKACCOUNT_CALLER`. If no account checks are required by the operation, the special constant `CHECKACCOUNT_NONE` can be passed.

The account status check is scheduled in the EVC, which will be performed after all batched operations have been completed.

- **Vault checks**

  Operations schedule a *Vault Status Check* to be performed. The vault check ensures that the borrow and supply cap are not exceeded, by comparing the current values with the ones saved during the snapshot, and updates the interest rate by querying the Interest Rate Model.

2.2.11 Method dispatch

Vaults implement a large number of methods, that do not fit when compiled into the bytecode size limit of a single contract. Contract *EVault*, implementing the main functionality of Euler Vault Kit, addresses the issue in a novel way by using the `use(module)` and `useView(module)` modifiers defined in `Dispatch.sol` to offload some of the code size into immutable implementation contracts.

The functionality of *EVault* is subdivided into 8 modules, where the functionality is implemented:

- **Token** which implements the functions for token transfers and approvals.
- **Vault** which implements liquidity provision.
- **Borrowing** which implements the methods to borrow from the vault and repay.
- **Liquidation** which implements the liquidation functionality.
- **Governance** which implements functionality to set vault parameters for the governance and to redeem the fees earned.
- **RiskManager** which implements vault and account status checks, and interest rate updates.
- **BalanceForwarder**, where the user functionality to enable and disable the balance forwarder is implemented.
- **Initialize**, which implements the initialization function called after vault deployment.

The *EVault* contract derives from all of these modules. If there was no code limit in the EVM, this would be enough to expose all the functionality to users. However, the EVM bytecode size limit prevents the contract to be deployed as-is. For this reason, the modules are also deployed individually as stateless implementation contracts, and in the *EVault* contract only some methods are included directly, while others are called by delegate-calling into the module contracts. Methods that are defined in `EVault.sol` decorated with `use(module)` or `useView(module)` decorator perform a delegate call to the specified module, forwarding the payload to the module. `useView(module)` in particular is rather complex, as solidity prevents a delegate call in a `view` method. Therefore, `useView()` first performs a `staticcall` to the `viewDelegate()` function of EVault, appending the original caller to the calldata so that it is retrievable, and `viewDelegate()` (which is not a view function) then performs the delegate call to the module.

2.2.12 Vaults Deployment

Vaults are deployed in a permissionless way from the *GenericFactory* singleton contract through the `createProxy()` method. `createProxy()` either deploys an upgradeable *BeaconProxy*, where the admin of the *GenericFactory* can set a new implementation for the *EVault* contract, or as a EIP-3448 immutable MetaProxy, which delegate calls to a fixed implementation contract for the EVault, which is the one currently set as implementation contract in the *GenericFactory*. Deployment with proxies allows upgradeability in the case of the *BeaconProxy* and reduces the gas cost of deploying a new vault. The proxies store basic vault information as immutable data, which includes the underlying asset of the vault, the price oracle for the vault, and the token used as the base unit for the price oracle. This data is forwarded in the *delegatecall* to the implementation contract by appending it to the calldata supplied by the user.

The *GenericFactory* deploys the proxy through the *CREATE* opcode, therefore the address of the newly deployed vault only depends on the nonce and address of the factory.

2.2.13 Synths

Synths are a product developed using Euler Vault Kit that implements a stablecoin system. *ESynth* is the stablecoin that is lent out by the system by Synth vaults. Synth vaults are a type of *EVault* where only governance is allowed to deposit *ESynth*, and users can obtain *ESynth* by borrowing it through the usual mechanism of Euler Vault Kit. The other specificity of Synth vaults is that they do not use the default interest rate model, but the *IRMSynth* interest model, which sets the interest rate depending on the price of *ESynth* compared to a reference asset. If the *ESynth* price is below that of the reference asset, the interest rate increases. Borrowers are incentivized by the high interest rate to repay their loans. To do so, they need to acquire *ESynth* on the market, pushing the price up, to provide it back to Synth vaults. Conversely, if the *ESynth* price is high compared to the reference asset, the interest rate is lowered, so that it will become beneficial for arbitrageurs to increase the supply of *ESynth* by borrowing more and exchanging it for the reference asset. A secondary peg-keeping mechanism for the shorter term is based on the *PegStabilityModule*. In the PSM, ESynth can be exchanged one to one for the reference asset and vice-versa, at a fixed fee. Arbitrageurs will use the *PegStabilityModule* to obtain a risk-free profit when the price depegs, and will, in turn, drive the price toward the peg.

ESynth can be staked in the *EulerSavingsRate* contract, a ERC-4626 vault where it will earn an interest. Depositors put *ESynth* in the ESR, and obtain shares of the vault in exchange. The interest comes from *ESynth* rewards deposited in the *EulerSavingsRate* by governance, which is distributed in a delayed way among ESR depositors, increasing the value of their ESR shares.

2.2.14 Roles and Trust Model

`EVault` defines a governor account. The governor is initially set to the creator of the vault proxy but can transfer his role to any other address. The governor can furthermore:

- Set the symbol and the name of the vault
- Set supply and borrow caps
- Set or unset a collateral LTV value
- Set the interest fee
- Set interest rate model
- Set the address of the fee receiver
- Set hooks
- Set config flags

Since the governor can set a collateral's LTV value or set hooks that can prevent specific users from redeeming their shares it must be fully trusted by the users.

`GenericFactory` is deployed by Euler. It defines an `upgradeAdmin` role. This role is trusted since it can set the implementation address that proxies use.

`ProtocolConfig` is also deployed by Euler. It defines an `admin` set at the creation of the contract and a `feeReceiver`. The `admin` can use `setFeeReceiver` to select the address that will receive the protocol fee shares from each deployed `EVault`. The `protocolFeeShare` taken by Euler can be set by `admin` with `setProtocolFeeShare`. The `admin` can also choose to give his role to another address.

`ESynth` is owned by the deployer of the contract. The `owner` can define how much synthetic asset each account can mint and can `allocate` or `deallocate` the synthetic asset to/from vaults. Last, `owner` can add or remove any address from being taken into account when calculating the total supply of the synthetic asset. The `owner` is fully trusted, as it has the power to mint `ESynth` assets.

Tokens used by vaults behave according to the ERC-20 standard, are non-rebasing and do not implement fees on transfers. Collaterals enabled in vaults are fully trusted, they are assumed to have been reviewed for compatibility with the EVault.

2.2.15 Changes in Version 2

The following changes have been made in Version 2 of the contracts:

- Collaterals now have distinct `borrowLTV` and `liquidationLTV` values instead of a single `LTV`. Both can be set by Governance with the restriction that the `borrowLTV` must be smaller or equal to the `liquidationLTV`. The `liquidationLTV` is only used during liquidations whereas the `borrowLTV` is used for all other liquidity checks which creates a distinction between the allowed collateralization ratio for borrowing and liquidation.
- A `liquidationCoolOffTime` period has been added in which a violator cannot be liquidated. This period can be configured by Governance and begins after the last successful account status check passed by the violator. The goal of this period is to prevent the violator from being liquidated in the same block as he borrowed. Even if `liquidationCoolOffTime` is set to 0, the violator can only be liquidated in the next block after the borrow and the resulting successful status check.
- In Version 1, the liquidation discount received by a liquidator was capped at 20%. Version 2 removes this fixed cap and instead allows Governance to set a custom maximum liquidation discount between 0% and 100%.
- `loop()` and `deloop()` have been removed from the `Borrowing` module and `repayWithShares()` has been added, which has the same functionality as `deloop()`.
- `OP_MAX_VALUE` and `CFG_MAX_VALUE` have been added to respectively delimit the number of possible operations and config flags.
- The governor can now also be authenticated if he calls the vault through the EVC. `EVCAuthenticateGovernor()` has been added to verify that the governor uses his owner account and not a subaccount, and that the caller is the governor himself and not an operator of the governor's account. Furthermore, the governor will not be authenticated while the `controlCollateralInProgress` flag in the EVC is set. The governor can however still be authenticated by calling the vault directly as it was the case in Version 1.
- Several gas optimizations have been made in the code.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- ESynth Interest Rate Can Be Manipulated Into Big Jumps Risk Accepted

Low-Severity Findings	5

- 1-Wei Repay Can Make Actual Repay Revert Risk Accepted
- Deployed Vaults Will Have the Same Address on Different Chains Risk Accepted
- EVault decimals() Can Return Different Values Depending on Gas Available Risk Accepted
- Interest Fee Shares Amount Does Not Use VIRTUAL_DEPOSIT_AMOUNT in initVaultCache() Risk Accepted
- Multiple Loans Can Bypass Oracles Liquidity Measure Acknowledged

5.1 ESynth Interest Rate Can Be Manipulated Into Big Jumps

Design Medium Version 1 Risk Accepted

CS-EVK-001

In `IRMSynth`, if the price of `ESynth` is below the reference, the interest rate can be increased by calling `computeInterestRate()`. However, it is not updated in the vault, unless an operation is performed on the vault.

Therefore what can happen is that `computeInterestRate()` is called repeatedly directly on the `IRMSynth`, for example for 20 consecutive hours of Vault inactivity. Now the `IRMSynth` has an interest rate that is 6x higher than before ($1.1^{20} - 1$), but this has no effect yet on the outstanding borrows because the new interest rate is not yet applied, so even if the interest rate is 6x higher there's no incentive for borrowers to repay because the old interest rate still applies.

When an operation happens on the vault, the interest rate will then suddenly increase by 6x, and will take at least another 20 hours to return to normal. This means every period of Vault inactivity can be exploited to exponentially increase or decrease the interest rate, potentially at the advantage or detriment of borrowers.

Risk accepted:

Euler accepts the risk with the following statement:

> We acknowledge this issue and for now, will accept the risk. If a synth vault often has long periods of inactivity, it is probably mostly a lower-tier, underused vault and the effect on interest is less important. Depositors (or borrowers) can always periodically call `touch()` on the vault too, to force an interest rate update.
>
> Since synths are typically governed/managed projects, the operator will typically retain governance control over the synth vaults and therefore have the ability to reconfigure the IRM. If this becomes a problem we will make a new version of the IRMSynth code, possibly by making a per-vault cache and authenticating the caller of `computeInterestRate()`.

5.2 1-Wei Repay Can Make Actual Repay Revert

Design | Low | Version 1 | Risk Accepted

CS-EVK-003

Anybody can call `repay()` to repay part of a position's debt. If a legitimate repay with a non `uint256.max` amount is frontrun with a small repay, for example of 1-wei, the legitimate repay can revert, if the remaining debt is smaller than the repay amount.

This can be a denial of service vector when the full repay amount is specified. A mitigation is to always use `uint256.max` when repaying the whole debt.

Risk accepted:

Euler accepts the risk with the following statement:

> We accept the report's conclusion, that the mitigation is to always use uint256.max when repaying the full loan. The recommendation does not require any code changes.

5.3 Deployed Vaults Will Have the Same Address on Different Chains

Security | Low | Version 1 | Risk Accepted

CS-EVK-005

`GenericFactory.createProxy()` either deploys a `BeaconProxy` or a `MetaProxy`. Both proxies are deployed using `create`. The address of a contract deployed through `create` depends on the nonce of the address that deploys it and on the sender address, which in this case is the `GenericFactory`. The starting nonce of the `GenericFactory` contract that deploys the proxy will be 0 on all chains.

If the `GenericFactory` is deployed at the same address on different chains, then the addresses of the deployed vault proxies will follow the same sequence on different chains. This fact can be exploited by malicious actors to deploy a vault similar to an existing one on a different chain, at the same address, and trick users into interacting with it. The deployed vault could use a malicious oracle for example and its governance would be not trusted.

Risk accepted:

Euler accepts the risk with the following statement:

> We will make sure that GenericFactory is deployed at different addresses for each chain.

5.4 EVault `decimals()` Can Return Different Values Depending on Gas Available

Security | Low | Version 1 | Risk Accepted

CS-EVK-006

The `Token.sol` function `decimals()` returns 18 if the `asset.decimals()` revert, and the returned value otherwise.

```
(bool success, bytes memory data) = address(asset).staticcall(abi.encodeCall(IERC20.decimals, ()));
return success && data.length >= 32 ? abi.decode(data, (uint8)) : 18;
```

`EVault.decimals()` can be called with a low gas amount, of which `63/64` are forwarded to the `staticcall` to asset. If this gas amount is not enough for the subcall to succeed, but the remaining 1/64 is enough for `EVault.decimals()` to return, 18 decimals will be returned even if the underlying asset implements a valid `decimals()` method that returns another value than 18.

Risk accepted:

Euler accepts the risk with the following statement:

> The EVault's code is agnostic of the asset's decimals, so the function is only relevant for external systems calling EVault. Also, given that the `decimals` function consumes presumably only small amounts of gas, the 1/64 that would be left to take advantage of the incorrect value would be minuscule and most probably insufficient for any state changes.

5.5 Interest Fee Shares Amount Does Not Use `VIRTUAL_DEPOSIT_AMOUNT` in `initVaultCache()`

Correctness | Low | Version 1 | Risk Accepted

CS-EVK-007

In `Cache.initVaultCache()`, `newTotalShares` represents the new total amount of shares in the vault after new shares have been minted for fees. It is only computed if there is an interest fee. However, the computation does not use `VIRTUAL_DEPOSIT_AMOUNT` which is used in all other conversions of shares to assets or vice versa. This leads to the number of fee shares being underestimated or overestimated depending on the vault's health and the `feeAssets` accrued.

In the following, we denote the number of shares per asset as k. If the vault is healthy, $k <= 1$ and if the vault has bad debt we have $k > 1$.

1. For $k >= 1$ the number of minted shares for fees will be slightly more if `VIRTUAL_DEPOSIT_AMOUNT` is not used in the calculation of `newTotalShares` and the total amount of assets in the vault is small. However, the difference becomes negligible when the number of assets in the vault increases or when $feeAssets > 10^6$
2. For $k < 1$ the number of minted shares for fees will be slightly less if `VIRTUAL_DEPOSIT_AMOUNT` is not used in the calculation of `newTotalShares` and the total amount of assets in the vault is small. However, the difference becomes negligible when the number of assets in the vault increases or when $feeAssets > 10^6$

This issue has only small security implications as in both cases the difference in fees is only noticeable if the amount of shares in the vault is close to 0. However, the calculation of `newTotalShares` is inconsistent with the vault's conversion standard from shares to assets and vice versa described in the whitepaper.

Risk accepted:

Euler accepts the risk with the following statement:

> We agree that the impact is negligible and prefer to keep the behavior as is. Comments in the relevant code were added to explain the effect.

5.6 Multiple Loans Can Bypass Oracles Liquidity Measure

Design | Low | Version 1 | Acknowledged

CS-EVK-008

Oracles return collateral and liability values which can take into account the market liquidity, according to the documentation, but this can be bypassed by creating multiple small loans instead of a single big one.

Oracles are passed the amount of token to convert, which can be used to return an output amount that depends on the liquidity of the market, that is, they can take account of the slippage that will result from selling the collateral, or acquiring the liability, when a liquidation happens. Indeed the oracles documentation (`IPriceOracle.sol` in the Euler price oracle repo) mentions:

```
/// @return bidOutAmount The amount of `quote` you would get for selling `inAmount` of `base`.
/// @return askOutAmount The amount of `quote` you would spend for buying `inAmount` of `base`.
```

This in principle can result in safer loan creation, since bigger loans can require a bigger collateralization ratio because the price takes into account slippage. However, this mechanism can be bypassed by simply creating multiple smaller loans instead of a single big one, if the oracle implements this mechanism.

Acknowledged:

Euler acknowledges the risk with the following statement:

> Currently, none of our oracles take into account trade size when performing the conversion. But the observation in this issue is well-noted and we will consider it if we do decide to build such oracles in the future.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Multiple Partial Liquidations Can Result in Higher Than Expected Discount Code Corrected

Low-Severity Findings	0
Informational Findings	6

- deposit() and redeem() Are Not ERC-4626 Compatible Specification Changed
- Inaccurate Comments or Lack of Documentation Code Corrected
- Event Indexing Code Corrected
- Gas Optimizations Code Corrected
- Synths Specification Does Not Include skim as a Vault Function to Be Disabled Specification Changed
- OP_VAULT_STATUS_CHECK Is Not Listed as a Controller Neutral Operation Code Corrected

6.1 Multiple Partial Liquidations Can Result in Higher Than Expected Discount

Design Medium Version 1 Code Corrected

CS-EVK-002

The dutch auction liquidation mechanism relies on a discount that progressively increases as the health of a position decreases. Some liquidations can worsen the health of the position instead of improving it, this makes it profitable for liquidators to split a single liquidation into smaller ones which progressively decrease the health, therefore increasing the discount at which collateral is acquired, resulting in a higher discount for liquidators than expected.

The health of a position is defined as (refer to System Overview):

$$h = \frac{\sum_{i=1}^{k} c_i v_i}{d}$$

where d is the debt value, and c_i is the i-th collateral value and v_i the i-th collateral's LTV. Values are all computed in terms of the same reference asset.

The discount factor applied to liquidations, when the health is below 1, is:

$$D = \max(h, 1 - MAX\_LIQUIDATION\_DISCOUNT)$$

where `MAX_LIQUIDATION_DISCOUNT` is defined as 20%. When repaying a debt amount of value d, an amount of collateral of value $\frac{d}{D}$ is paid to the liquidator. Since $\frac{d}{D}$ is greater than d, the liquidator has an incentive to perform the liquidation.

However, partial liquidations are possible and can result in a position's health which is either better or worse than the starting one. If a partial liquidation improves the health of the position, the collateralization improves, which reduces the risk of bad debt for the system, and the position can either become healthy again, preventing further liquidations, or the liquidation discount decreases making further liquidations less detrimental for the borrower. On the other hand, if a partial liquidation worsens the position, the liquidation discount will increase, making further liquidations even more profitable for liquidators, and possibly accumulating more bad debt for the system.

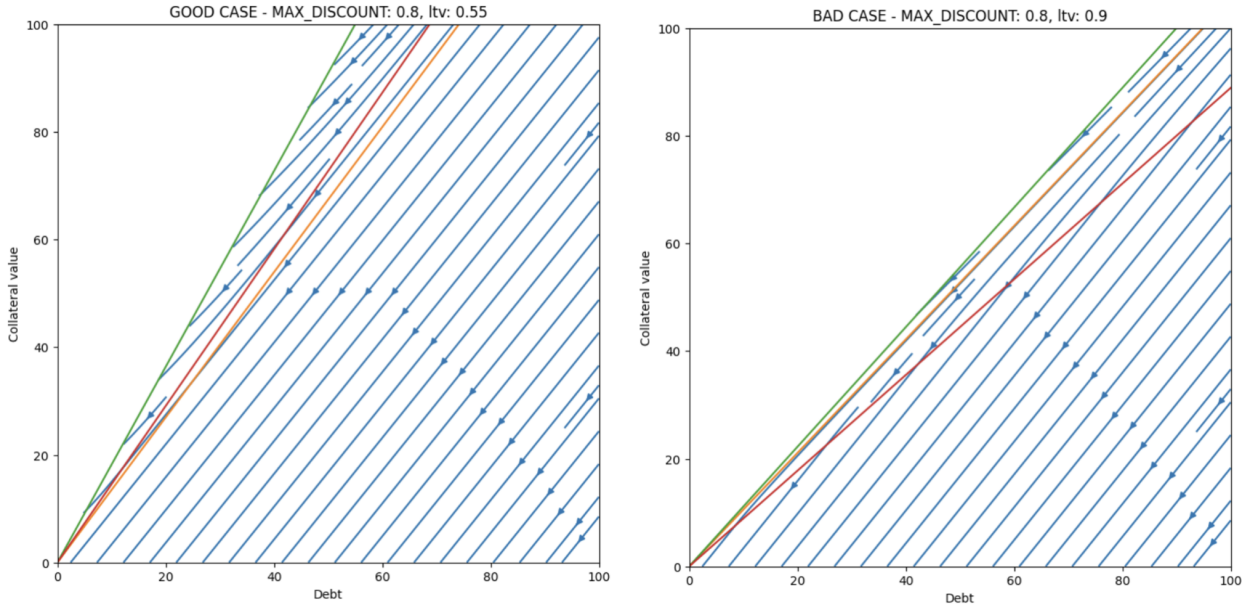
Let's for example consider a position where the borrower owes 80 units of debt, has 85 units of collateral and the LTV of the collateral is 80%. The health of the position is $\frac{85 \times 80\%}{80} = 85\%$, which means the current discount is 15%.

If the liquidator chooses to perform a single big liquidation, they can repay 72.25 units of debt, and receive the whole position collateral $\frac{72.25}{0.85} = 85$. The remaining 7.75 units of debt will become bad debt for the vault, since they don't have any more collateral to back them. The liquidator will have paid 72.25 to obtain 85 worth of collateral, realizing a profit of 12.25.

The problem lies in the liquidator instead choosing to perform multiple liquidations, with the same starting state. If they first perform a liquidation worth half the collateral, by repaying 36.125 and obtaining $\frac{36.125}{0.85} = 42.5$, the new state of the system will be a debt of 43.875 and a collateral of 42.5. The health of the system is now $\frac{42.5 * 80\%}{43.875} = 0.775$, and the discount factor `D` is therefore 80% (the lower bound kicks in) instead of the previous 85%. The liquidator can perform a second liquidation by repaying 34 units of debt and obtain the remaining 42.5 units of collateral. The remaining unbacked debt will be 9.875, and the liquidator's profit 14.875, which is higher than in the single liquidation case. Liquidators can therefore realize higher profits than expected by splitting liquidations that decrease the position's health into multiple smaller liquidations.

It can be shown that vaults with collateral LTV higher than 64%, which is the square of the minimum *discount factor* 80%, have states where it is preferable for liquidators to perform multiple liquidations. This is because when the health is below the square root of the LTV, a liquidation will further decrease the health.

The following two plots illustrate the liquidation dynamics when liquidations cannot decrease the health (good case), and when liquidations can decrease the health (bad case). The blue lines show the evolution of the position as infinitesimal liquidations are performed. The green line demarcates the transition between a healthy position where liquidations are not possible, and an unhealthy one where liquidations can be done. The red line is the limit below which the maximum discount of 20% kicks in, so all blue lines below this line are parallel and apply the max liquidation discount. The orange line defines two regions, above the line, liquidations improve the position, and below the line, they make it worse. If the orange line is below the red line, the maximum discount has already kicked in, and multiple liquidations cannot improve the discount further. But if the orange line is above the red line, then the region between the orange and the red line represents the states for which the position can be made more unhealthy by multiple liquidations at a profit for liquidators.



In the bad case, where the LTV is 90%, the orange line marks the states where the health is at $\sqrt{0.9} = 0.949$. The blue flow lines located around the orange line show that the system is moved toward a healthier state when the starting point is above the orange line, or toward the unhealthier max discount (red line) when starting from below the orange line. When the state is just below the orange line, a liquidator can start performing multiple small liquidations and instead of the starting discount of ~5% ($1 - \sqrt{0.9}$), quickly reach the maximum discount of 20%.

Code Corrected:

Euler modified the liquidation system such that every vault has now a configurable maximum discount factor. Governors have to select an appropriate maximum discount based on the highest configured LTV of the vault. It is the responsability of the Governor to provide an appropriate value.

6.2 Event Indexing

Informational | Version 1 | Code Corrected

CS-EVK-012

In `Events`, `from` and `to` addresses are not indexed in the `PullDebt` event.

Code changed

Event was updated to index the addresses.

6.3 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-EVK-013

- In `Liquidation`, `liabilityValue` is computed using `liqCache.liability.toUint()`. However, there is no need to compute the liability value again as previously a call to `calculateLiquidity()` is made with `liquidation` set to true which returns `liquidityLiabilityValue` as second return value. This value can directly be used as `liabilityValue`. This will save gas as it avoids recomputation involving the oracle.

- In `BalanceUtils`, `decreaseAllowance()` reads the spender allowance even if `spender == owner`. This storage load can be avoided, and would result in gas save for every transfer, since `decreaseAllowance()` is always used in transfers.

Code changed

Both optimizations were implemented as recommended.

6.4 Inaccurate Comments or Lack of Documentation

Informational | Version 1 | Code Corrected

CS-EVK-014

- In `Liquidation.sol` the comment about limiting the yield to the borrower's available collateral reads:

```
// Limit yield to borrower's available collateral, and reduce repay if necessary
// This can happen when borrower has multiple collaterals and seizing all of this one won't bring the violator back to solvency
```

However, this can also happen when the borrower only has one collateral and seizing it won't bring the violator back to solvency.

- In `ConfigAmount.sol` there is a comment that states:

```
// ConfigAmounts are floating point values encoded in 16 bits with a 1e4 precision.
```

This comment can lead to confusion as the `ConfigAmounts` values are not floating point values, but fixed point values.

- In `RiskManager.sol`, at line 68, `disableCollateral()` is mentioned as an *EVault* function that does not change the *EVault* storage. `disableCollateral()` is not an *EVault* method, what is probably meant here is `disableController()`.

Code corrected:

Comments were corrected as recommended.

6.5 Synths Specification Does Not Include `skim` as a Vault Function to Be Disabled

Informational | Version 1 | Specification Changed

CS-EVK-016

The specification for Synths vault stipulates that "Euler synthetic vaults are a special configuration of vaults which use hooks to disable deposit, mint and loop for all addresses [...] disallowing deposits for normal users." However, the specification does not include the "skim" function as a vault function to be disabled whereas it allows a normal user to deposit funds into the vault.

Specification changed:

Synths specification was updated to include `skim()`.

6.6 `OP_VAULT_STATUS_CHECK` Is Not Listed as a Controller Neutral Operation

Informational **Version 1** **Code Corrected**

CS-EVK-018

`Constants` defines `CONTROLLER_NEUTRAL_OPS` which are operations that do not require the account to have the current vault installed as a controller. However, the `OP_VAULT_STATUS_CHECK` operation is not included in this list.

`CONTROLLER_NEUTRAL_OPS` are used in `Base.initOperation()` to decide whether or not to check if the vault is installed as a controller for the calling account. As `initOperation` is not called with `OP_VAULT_STATUS_CHECK`, it is not an issue that `OP_VAULT_STATUS_CHECK` is not included in `CONTROLLER_NEUTRAL_OPS` but simply an inconsistency.

Code changed:

Operation was added to the list of controller neutral operations.

6.7 `deposit()` and `redeem()` Are Not ERC-4626 Compatible

Informational **Version 1** **Specification Changed**

CS-EVK-020

In `Vault.sol`, `deposit()` and `redeem()` will convert an `amount` of `type(uint256).max` to the maximum amount of funds the user has available. This behavior is not ERC-4626 compatible. While the EVK white paper provides a list of the EVK features that are ERC-4626 compatible, it does not list the above behavior as being an existing incompatibility.

Specification changed:

The white paper was updated to list this incompatibility.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Rounding Errors

Informational | Version 1 | Risk Accepted

CS-EVK-015

Several minor rounding issues that can occur:

- In `Liquidations.calculateLiquidation()`, `liqCache.liability` is the violators debt rounded up. However, the liability value is used to compute how much collateral the liquidator will receive when liquidating the violator. Therefore, if the liability of the violator is 1.1 wei (1 wie + dust), it will round to 2. The liquidator's reward will then be computed as if he took over 2 wei of debt, however the debt actually transferred to the liquidator (`transferBorrow()`) will only be 1.1.
- In `ConversionHelpers.conversionTotals()` sub-wei interest earned in `totalBorrows` is always rounded up using `toAssetUp()`. This slightly overestimates Vault's assets, and can lead to rounding in the wrong direction in `Shares.toAssetsDown()` and `Assets.toSharesDown()`. For example, if total assets are 10^6 (virtual assets) + 1.1 (partial interest accrued), and the shares amount is 10^6 (virtual shares) + 2 then `toAssetsDown(1)` will return 1 instead of 0. Since `Shares.toAssetsDown()` is used in `redeem()`, it allows any user to redeem 1 wei more if for example amount to redeem is 1e6, amount of shares is 2e6 and amount of assets is (2e6-0.99). The user should receive 1e6 - 1 but will get 1e6 assets.

Risk accepted:

Euler accepts the risk with the following statement:

> The liquidation yield and repay calculation follows an arbitrary algorithm and the exact values are much more dependent on transient market conditions, than rounding nuances.
>
> As for rounding the borrows up in total assets, we believe it is correct, because there is no possibility of repaying partial, sub-wei debt. Indeed internal debt precision is not observable for the users borrowing and repaying. The sub-wei debt is therefore equivalent to a full wei debt, because it would require a full wei to remove it.

7.2 `IRMSynth` Is Subject to Oscillating Interest Rate

Informational | Version 1 | Risk Accepted

CS-EVK-017

`IRMSynth` adjusts the interest rate for borrowing synthetic assets based on the price of the synthetic asset relative to the reference asset (peg). The interest rate will still be adjusted if the price of the ESynth is exactly equal to target price (`TARGET_QUOTE`). Therefore, the interest rate can never be constant but always oscillates. This oscillation happens over 1 hour periods.

Risk accepted:

Euler accepts the risk with the following statement:

> We acknowledge this issue and for now will accept the risk. Since synths are typically managed products, the governor will usually retain governance control over the synth vaults. We will monitor this and if it becomes an issue, we will create a new IRMSynth that addresses this, possibly by creating a “goldilocks” zone where no adjustments are made.

7.3 `ProxyUtils.metadata()` Return Arbitrary Values When Implementation Contract Is Called Directly

Informational **Version 1** **Risk Accepted**

CS-EVK-019

If a user calls the EVault implementation directly and not through a proxy, the addresses returned by `ProxyUtils.metadata()` can be set arbitrarily, since the data is entirely in the control of the user. This can result into external calls to user controlled contracts.

The subtractions in `metadata()` can also result in an underflow if the user does not provide enough calldata.

The implementation expects the metadata to be at the end of the calldata and will always try to read at least 60 bytes from the end of the calldata. If the calldata is shorter than 60 bytes, an underflow can occur.

```
assembly {
    asset := shr(96, calldataload(sub(calldatasize(), 60)))
    oracle := shr(96, calldataload(sub(calldatasize(), 40)))
    unitOfAccount := shr(96, calldataload(sub(calldatasize(), 20)))
}
```

A problem that could emerge from being able to set arbitrary addresses in the implementation contract directly, on chains where `selfdestruct` is not disabled, is being able to trigger a `delegatecall` to an arbitrary address. The only `delegatecall` to a non immutable address in *EVault* is in `viewDelegate()`. `viewDelegate()` can only be called from the contract itself. Since no external call performed by the contract collides with the `viewDelegate()` selector, it is not possible to replace the address of a contract used by *EVault* to trick the implementation contract into a call to `viewDelegate()` with arbitrary payload.

Risk accepted:

Euler accepts the risk with the following statement:

We agree with the analysis that it is not possible to use the described method to make an arbitrary delegate call and potentially execute selfdestruct on chains which support it

7.4 `interestRate()` View Function Can Be Manipulated

Informational **Version 1** **Acknowledged**

The `interestRate()` view function of `Borrowing` can be manipulated without any capital cost, when it is used within an EVC batch. If external systems can be triggered to read `interestRate()`, they could be made to read an inflated value. The attacker would make an EVC batch that:

1. borrows all liquidity in the vault.
2. triggers the external system that reads `interestRate()`.
3. repay the borrow.

The stored interest rate of the vault is more robust to manipulation, as it can't be inflated by this attack or by flashloans, but has to be manipulated using actual capital.

Acknowledged:

Euler acknowledges this behavior with the following statement:

> This behavior is by design so that simulations can read the effect of interest rate changes mid-batch. Note that this attack is possible even without EVC batches, by using flash loans from external systems. Contracts should generally not be reading instantaneous interest rates not only because they can be manipulated, but also because even without manipulation they won’t take effect until the next block and can be changed by subsequent operations in the same block.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Borrow Amount Is Rounded-Down When Applying Interest Accumulator

Note **Version 1**

In `Cache.initVaultCache`, `newTotalBorrows` is computed in the following way:

```
uint256 newTotalBorrows = vaultCache.totalBorrows.toUint() * newInterestAccumulator / vaultCache.interestAccumulator;
```

We note that the division will round down the result. Therefore, if the interest rate is small enough, interest rate will not accrue as `newTotalBorrows` will be rounded down to `vaultCache.totalBorrows`.

8.2 LTV of a Collateral Can Be Decreased With `rampDuration` 0

Note **Version 1**

In `setLTV()`, the governor can decrease the LTV of a collateral using a `rampDuration` which allows for a linear LTV decrease over the given duration to the new target LTV instead of an immediate change. This gives users time to adjust their debt positions to avoid liquidation. However, `rampDuration` can be set by 0 by the governor during an LTV decrease.

8.3 New Fee Distribution Is Applied Retroactively in `Governance.convertFees()`

Note **Version 1**

In `Governance.convertFees()`, if a new `protocolFeeShare` has been set with `ProtocolConfig.setProtocolFeeShare()` and `convertFees()` has not been called yet, the new protocol fee distribution will be applied retroactively to previous unclaimed fees, taking a possibly bigger share of the fees earned by the vault.

8.4 Optional Flashloan Fee Can Be Avoided

Note **Version 1**

CS-EVK-009

The whitepaper mentions implementing a flashloan fee through the `OP_FLASHLOAN` hook. The hypothetical fee can be avoided by performing an EVC batch containing an initial `borrow()` and a final `repay()`, which is functionally equivalent to a flashloan. Vault governors wishing to enable flashloan fees through hooks should consider charging an initiation fee on regular borrows as well.

8.5 Violator Can Enable More Collateral to Increase Liquidation Cost

Note Version 1

In `LiquidityUtils.checkLiquidity()`, the total collateral value of the violator is computed as the sum of the collateral value of his different collaterals. A violator can on purpose enable the maximum amount of collaterals (which is set to 10 in the Ethereum Vault Connector) such that the liquidation cost is increased as the liquidation process iterates over all collaterals to compute the total collateral value. Invalid collaterals (which are not enabled by the vault) are read from the EVC storage but skipped in the liquidity check, however they each increase the gas consumption of the liquidity check by around 5000 gas, dominated by the SSLOAD cost in the EVC contract to read the address of the collateral, and the SSLOAD cost in `LiquidityUtils` to read the collateral's LTV (`getLTV()`) (which will be unset if the collateral is not enabled).

8.6 `IRMSynth` Rate Will Take Time to Converge After Initialization

Note Version 1

The interest rate in `IRMSynth` is initialized to BASE_RATE which is defined as 0.5% APR. As the rate can only be adjusted by a maximum of 10% of the current rate every hour, it may take some time for the rate to be effective in the peg keeping mechanism.

8.7 `gulp` Can Distribute Interest Over More Time Than `INTEREST_SMEAR`

Note Version 1

`EulerSavingsRate.gulp()` distributes interest that has not been yet accounted for over 2 weeks after it has been invoked. However, every time `gulp()` is called, the interest that has not been distributed yet is smeared again over 2 weeks from the current block timestamp.

As a simple example, assume we are distributing 1000 underlying assets over `INTEREST_SMEAR` period (2 weeks), then the distribution rate will be `1000 * 10^18 / 14 days` assuming that the underlying asset has 18 decimals.

The amount of interest distributed after 1 week will be:

$$distributed_{7days} = 1000 * 10^{18} * 7days/14days = 500 * 10^{18}$$

If `gulp()` is called again after 1 week but no new interest has been deposited, then the distribution rate will be `500 * 10^18 / 14 days`. Thus, the interest distribution will last 1.5 times the expected duration with the last 14 days at half the rate.

Let's now suppose that `gulp()` is called every block and no new interest gets deposited apart from an initial amount of 1000 underlying assets to be distributed. The interest rate distribution will be as follows:

$$distributionRate_t = \frac{(distribution_{t-1} - distributionRate_{t-1} * blockDuration)}{distributionDuration}$$

$$= \frac{distribution_{t-1} - \frac{distribution_{t-1}}{RewardDuration} * blockDuration}{distributionDuration}$$

$$= \frac{distribution_{t-1}}{distributionDuration} * (1 - \frac{blockDuration}{distributionDuration})$$

$$= distributionRate_{t-1} * (1 - \frac{blockDuration}{distributionDuration})$$

Over `n` blocks the `distributionRate` will decrease from the initial `distributionRate` by `(1-blockDuration/distributionDuration)^n`, which is an exponential decay for the `distributionRate`, corresponding to an exponential decay of the remaining interest to be distributed. The interest will therefore not be distributed in a finite amount of time. Numerical simulations have shown that after 2 weeks 63% will have been distributed, after 3 weeks 78%, after 4 weeks 87%, after 6 weeks 95%.

Anybody can call `gulp()` at every block. However, the cost of doing so in terms of gas is likely to offset any advantage that such an attacker can get from delaying the distribution of the interest.

Furthermore, interest is expected to be deposited regularly, which makes delaying the distribution of interest even less impactful as the rate of distribution will reach a steady state.