SPEARBIT

Euler Labs - EVK Security Review

Auditors

Christoph Michel, Lead Security Researcher

Emanuele Ricci, Lead Security Researcher

M4rio.eth, Security Researcher

Christos Pap, Associate Security Researcher

David Chaparro, Junior Security Researcher

Report prepared by: Lucas Goiriz

July 5, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Euler Labs is a team of developers and quantitative analysts building DeFi applications for the future of finance.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of euler-vault-kit according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 30 days in total, Euler Labs engaged with Spearbit to review the euler-vault-kit protocol. In this period of time a total of **103** issues were found.

Summary

Project Name	Euler Labs
Repository	euler-vault-kit
Commit	2bcd7e...c61d0d
Type of Project	DeFi
Audit Timeline	Apr 8 to May 10
Two week fix period	May 10 - May 20

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	7	5	2
Low Risk	24	9	15
Gas Optimizations	7	3	4
Informational	63	30	33
Total	**103**	**49**	**54**

5 Findings

5.1 High Risk

5.1.1 PegStabilityModule assumes underlying has 18 decimals

Severity: High Risk

Context: PegStabilityModule.sol#L77-L91

Description: The swap amounts in the PegStabilityModule are quoted in the same decimals as the synth, which is always 18. It is therefore only compatible with underlying's of 18 decimals. If the pegged underlying has different decimals, it's profitable to perform the swap.

Recommendation: Consider scaling the quoted amounts based on the involved tokens' assets.

Euler: Fixed in commit f97807d8.

Spearbit: The commit mitigate the issue. Euler should consider documenting the PegStabilityModule constructor with natspec docs to provide further explanation on which are the expected value to be used for the contract deployment.

Euler: Natspec added in commit d976be34.

5.1.2 Self-liquidations of leveraged positions can be profitable

Severity: High Risk

Context: Liquidation.sol#L127

Description: An attacker can perform the following attack by sandwiching a price oracle update:

1. Taking on a leveraged position by flashloaning collateral and max-borrowing the debt token.
2. Performing the price update.
3. Liquidating themself (from another subaccount).

Profitability: The attack is profitable when the entire collateral balance c_l is seized (to repay the flashloan) while repaying fewer debt assets than assets that were borrowed. This difference of maxBorrowAssets - maxRepayAssets of debt assets is the profit.

```
# discount factor: df = 1 - discount
# collateralPrice_0 = price before the oracle update
# collateralPrice_1 = price after the oracle update
collateralPrice_1 = collateralPrice_0 * (1 - priceDrop)

# the maximum debt asset we can borrow is
maxBorrowAssets = LTV_borrow * collateralBalance * collateralPrice_0 / debtPrice

# from the liquidation code we see that
seizedAssets
= repayValue / discountFactor / collateralPrice_1
= (repayAssets * debtPrice) / discountFactor / collateralPrice_1

# expressed in terms of repayAssets that seize the maximum (entire) collateral balance
maxRepayAssets = collateralBalance * discountFactor * collateralPrice_1 / debtPrice

# profitable if this inequality holds
maxBorrowAssets > maxRepayAssets
LTV_borrow * collateralBalance * collateralPrice_0 / debtPrice
> collateralBalance * discountFactor * collateralPrice_1 / debtPrice
<=> LTV_borrow > discountFactor * (1 - priceDrop)
```

The `discountFactor` is set to `max(hs_liquidation, 0.8)`. The attack is profitable if an attacker can sandwich a price oracle update that would end up with `LTV_borrow > discountFactor * (1 - priceDrop)`.

Some oracle adapters, like Redstone and Pyth, allow the users to update or even choose a preferable price. In this case, the attack could even be performed in a single transaction batch for risk-free profit.

> Note: Using several smaller liquidations can increase the overall liquidation discount and lead to a more profitable attack. A profitable attack also leaves bad debt for the protocol.
>
> See this Notebook for further profitability analysis.

Example: `LTV_borrow = LTV_liquidation = 90%`. Oracle quotes 1 collateral at $1 (and debt is fixed at $1). Sandwich collateral oracle price update to $0.90:

1. Flashloan `1000` collateral and build a position of `(1000 collateral, 900 debt)` at `LTV_borrow`.
2. Oracle sets collateral price to `$0.90`. (for example, Redstone / Pyth require the user to trigger the update.)
3. Liquidate self by repaying `maxRepayAssets = 810`.

```
discountFactor = healthscore_liquidation = collateralBalance * collateralPrice_1 * LTV_liquidation /
↪  debtValue = 0.90

maxBorrowAssets: 900
maxRepayAssets: 810
seizedAssets: maxRepayAssets * debtPrice / discountFactor / collateralPrice_1 = 810D * 1$/D / 0.9 /
↪  0.9$/C = 1000C
Profit: maxBorrowAssets - maxRepayAssets = 900 - 810 = 90
```

The current maximum discount is set to 20% which can lead to profitable attacks for high LTV collateral assets even for small price drops. The remaining debt will be bad debt for the protocol and might be socialized across all lenders.

Recommendation: Oracle frontrunning attacks can't be fully mitigated. However, to reduce the risk of such an attack (risk-free, in the same block or transaction) the discount and the LTV of the collateral assets play an important role. Furthermore, restricting the discount factor could lead to unprofitable liquidations and more debt by itself. Therefore, consider choosing the maximum discount factor based on the chosen LTV configurations. This parameter could be set by governance based on an acceptable attack risk vs. liquidation incentives tradeoff.

Oracles for collaterals with a price deviation update threshold larger than `1 - sqrt(LTV_borrow)` should be considered unsafe.

Euler: See PRs:

- EVK PR 191
- EVC PR 157

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" where 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 that 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 setup 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.

Spearbit: The PR mitigate the liquidation issue with the mix of configurable `maxLiquidationDiscount` and the `liquidationCoolOffTime` but we suggest some changes:

- natspec for `LTVBorrow` in `IEVault` is inaccurate: the borrow LTV parameter will not be used only when "originating a position" but also in any case that the vault will require an account status check like increasing the borrow position, decreasing the collateral position and so on. The natspec should be updated to cover all the scenarios.
- natspec for the `borrowLTV` parameter in the `setLTV` function inside `IEVault`: same comment as for the `LTVBorrow` natspec.
- natspec for the `setLiquidationCoolOffTime` function in `IEVault`: the natspec is wrong, the function is not a Getter but a Setter, it does not "Retrieves liquidation cool off time" but "**Set** liquidation cool off time".
- Consider adding to the `setLiquidationCoolOffTime` natspec the "side effects" of setting it to `0`: users will be able to be liquidated in the same block.
- The `setLiquidationCoolOffTime` has no upper bound sanity check. Consider adding a sane upper bound to prevent to not being able to liquidate a borrower for too much long.
- The `setMaxLiquidationDiscount` has no lower or explicit upper bound (the upper bound is `1e4` given the implicit check done by `.toConfigAmount()`). A `maxLiquidationDiscount` equal to `0` means that the liquidator will not get a bonus by repaying the violator's debt.
- Consider enforcing the `Governor` to set up the initial values of `maxLiquidationDiscount` and `liquidationCoolOffTime` when the `EVault` is initially deployed. Right now, there's nothing that prevents the `Governor`

to configuring the `LTV`s without having configured an initial value of `maxLiquidationDiscount` and `liquidationCoolOffTime` that will be both equal to zero:

  - `maxLiquidationDiscount == 0` means that the liquidator will not get any bonus for performing the liquidation.
  - `liquidationCoolOffTime == 0` means that the liquidator will be able to liquidate the users in the same block when the last account status check has been performed successfully.

- Consider renaming the `LTVConfig` struct attributes as following for a better clarity:
  - `liquidationLTV` → `targetLiquidationLTV`, to be explicit that this is not the current LTV (or at least it depends on ramp) and to make a statement that it's different compared to how `borrowLTV` is returned (it's always static value).
  - `targetTimestamp` → `targetLiquidationTimestamp` to be clear that it's not related to `borrowLTV`.
  - `rampDuration` → `targetLiquidationRampDuration` to be clear that it's not related to `borrowLTV`.

Euler:

> natspec for LTVBorrow in IEVault is inaccurate: the borrow LTV parameter will not be used only when "originating a position" but also in any case that the vault will require an account status check like increasing the borrow position, decreasing the collateral position and so on. The natspec should be updated to cover all the scenarios.

Fixed in commit d5408f30.

> natspec for the borrowLTV parameter in the setLTV function inside IEVault: same comment as for the LTVBorrow natspec

Fixed in commit d5408f30.

> natspec for the setLiquidationCoolOffTime function in IEVault: the natspec is wrong, the function is not a Getter but a Setter, it does not "Retrieves liquidation cool off time" but "Set liquidation cool off time"

Fixed in commit 91513d98.

> Consider adding to the setLiquidationCoolOffTime natspec the "side effects" of setting it to 0: users will be able to be liquidated in the same block.

Fixed in commit d5408f30.

For the rest of comments:

Acknowledged, no fix. The governor is considered trusted. It is not the vault's responsibility to judge whether the configuration is reasonable. Such opinions can be encapsulated either in external contracts, which can be granted admin privileges to enforce certain limits, or in off-chain filters.

5.2 Medium Risk

5.2.1 Interest accumulated, but not accounted yet, could be reset if `Governance` updates the interest rate model when in "Interest Overflows" state

Severity: Medium Risk

Context: Governance.sol#L243-L254, Cache.sol#L106-L115

Description: Let's assume that borrowers have borrowed an amount `X` for which when `Cache.loadVault()` is executed, the interest accrued added to `X` would make the `newTotalBorrows` value overflow the `MAX_SANE_DEBT_AMOUNT`.

When such a scenario happens, the `totalBorrows`, `interestAccumulator` and `lastInterestAccumulatorUpdate` are not updated (same for `accumulatedFees` and `totalShares`). In practice, the interest accrued is "paused" and won't be accrued and grow until enough will be paid to allow the next calculation of `newTotalBorrows` to not overflow anymore.

If during this scenario, the Governance updated the interest rate model to an empty one or a reverting one, all the accumulated interest will be reset and lost forever.

```
function setInterestRateModel(address newModel) public virtual nonReentrant governorOnly {
    VaultCache memory vaultCache = updateVault();

    vaultStorage.interestRateModel = newModel;
    vaultStorage.interestRate = 0;

    uint256 newInterestRate = computeInterestRate(vaultCache);

    logVaultStatus(vaultCache, newInterestRate);

    emit GovSetInterestRateModel(newModel);
}
```

1) `updateVault()` won't update the vault storage/cache because of the overflow.

2) `vaultStorage.interestRate = 0` update the interest rate to zero, meaning that no interest will be accrued anymore.

3) `computeInterestRate(vaultCache);` when the model reverts or is `address(0)` do not update `interestRate` leaving it to the previous value that in this case is `0` (update in the instruction above).

At this point, the next time `updateVault()` will be called it will not overflow anymore because the `interestRate` is zero and `newInterestAccumulator` is equal to `vaultCache.interestAccumulator` that has not been updated since the beginning of the overflow phase.

`totalBorrows` and `interestAccumulator` will be updated with the current values (no changes) and `lastInterestAccumulatorUpdate` will be updated to `block.timestamp`, resulting in a loss of the total interest accrued but never accounted until now (since the start of the interest overflow period).

Recommendation: Euler should consider documenting this scenario or evaluate the possible sanity checks to be performed when `Governance` updates the interest rate model during an "Interest Overflow" period.

Euler: Acknowledged, no fix. White paper was updated to better cover this and similar effects.

Spearbit: Acknowledged.

5.2.2 `Governance.setInterestRateModel` is missing sanity checks

Severity: Medium Risk

Context: Governance.sol#L242-L254

Description: The `Governance.setInterestRateModel` is not actively checking the user `newModel` input that represents the new IRM rate model. When a new `IRM` is provided, the interest rate is resetted to `0` and then updated via `computeInterestRate(vaultCache)`. The transaction should revert when:

- `newModel` is equal to the current model.
- `newModel` is a broken `IRM` model that will revert when `computeInterestRate` is executed.

The second case should be correctly handled, given that it violates a white paper invariant defined in the Interest Rate section:

> When a vault has address(0) installed as an IRM, an interest rate of 0% is assumed. If a call to the vault's IRM fails, the vault will ignore this failure and continue with the previous interest rate.

Because the interest rate has been already reset to `0`, when the new interest rate is called and reverts, it won't update the value to the old one but will remain equal to `0`. In general, this case should be handled because the governance should not be able to actively set the IRM to a faulty one. Allowing such case will mean that borrowers won't accrue any interest on their open position and lenders will not accrue any rewards.

To be able to handle this case, the `computeInterestRate` must be refactored to return if the `IRM` call has reverted.

Recommendation: Euler should prevent the governance from setting the new `IRM` model to the same one already used or to a faulty one that will revert when executed.

Euler: We acknowledge the issue. Governance is considered trusted. Even if a reverting IRM is installed, it will not be considered a malicious action, but a user error, and as such will be expected to be remedied promptly.

Spearbit: Acknowledged.

5.2.3 `EulerSavingsRate` `maxWithdraw` and `maxRedeem` are not returning the correct underestimated value when `owner` has a controller enabled

Severity: Medium Risk

Context: EulerSavingsRate.sol

Description: The `EulerSavingsRate` is an `ERC4626` vault that integrates with the `EVC` ecosystem, allowing the module to be used as collateral for `EVK` vaults.

Because ESR shares can be used as collaterals, it's important that operations like `transfer`, `transferFrom`, `withdraw` and `redeem` ensure that users who have enabled a controller are still healthy after the execution of such operation. For this reason, any of the above functions executes the `EVCUtil.requireAccountStatusCheck`.

Because of this integration with `EVC` and `EVK`, the `EulerSavingsRate` module must implement the same logic implemented by `Vault` when the `ERC4626` functions `maxRedeem` and `maxWithdraw` are called. The value returned by such functions should be underestimated to zero if the `owner` parameter has enabled a controller (the user could be unhealthy and the transaction could revert).

Recommendation: Euler should override the `maxWithdraw` and `maxRedeem` functions and return **zero** if `hasControllerEnabled(owner)` returns `true`.

Euler: Fixed in commit 49aaca39

Spearbit: The commit mitigate the issue. The logic has been well explained with the inline comments provided by the additional commit c479c4c5.

5.2.4 IRMSynth's targetQuote assumes reference asset has 18 decimals

Severity: Medium Risk

Context: IRMSynth.sol#L9

Description: The IRMSynth.targetQuote parameter is set to 1e18. It is compared against the output of oracle.getQuote(1e18, synth, referenceAsset) that returns a reference asset amount which will be in reference asset decimals.

Recommendation: Consider shifting the targetQuote based on the referenceAsset.decimals() in the constructor.

Euler: Fixed in commit affd8844.

Spearbit: The provided commit mitigate the issue. Spearbit suggest considering the following changes

- ESynth that inherits from ERC20Collateral does not allow having a custom value for decimals that will be by default equal to 18. If there's not a specific reason to have it as an arbitrary parameter, Euler should consider simplifying the code and declaring quoteAmount as a constant equal to 1e18
- There is no sanity check on the targetQuoute_ input parameter. Depending on the value passed, the IRM could for example always increase or decrease the rate no matter the quote value in _computeRate. Consider adding sanity checks or documenting it

Euler: Will keep as it is now to avoid possible integration issues with different synth tokens in the future. Will add to the docs that targetQuote should be properly setup.

5.2.5 maxMint/maxDeposit can overestimate shares/assets as it ignores totalShares overflow

Severity: Medium Risk

Context: Vault.sol#L64-L66

Description: The maxMint function currently returns shares < MAX_SANE_AMOUNT ? shares : MAX_SANE_AMOUNT where shares are the max-deposit assets converted to shares. However, it needs to take the current totalSupply into account as totalSupply + shares <= MAX_SANE_AMOUNT should hold.

It can return a larger amount than what can actually be accepted, according to EIP4626, this breaks the behavior:

> MUST return the maximum amount of shares mint would allow to be deposited to receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).
>
> Note that maxDepositInternal only looks at cash and if the vault has a high utilization, maxDepositInternal might return a large value, indeed resulting in a large shares amount that would overflow the totalSupply's MAX_SANE_AMOUNT.

Recommendation: Consider changing the maxMint and maxDeposit functions to take into account totalShares reaching the MAX_SANE_AMOUNT limit.

```
  // example for maxMint only
  function maxMint(address account) public view virtual nonReentrantView returns (uint256) {
      VaultCache memory vaultCache = loadVault();

      if (isOperationDisabled(vaultCache.hookedOps, OP_MINT)) return 0;

      // make sure to not revert on conversion
      uint256 shares = maxDepositInternal(vaultCache,
↪   account).toAssets().toSharesDownUint256(vaultCache);

-     return shares < MAX_SANE_AMOUNT ? shares : MAX_SANE_AMOUNT;
+     uint256 remainingSupply = vaultCache.totalShares - shares;
+     return shares < remainingSupply ? shares : remainingSupply;
  }
```

Euler: Fixed in PR 155.

Spearbit: The provided PR mitigates the issue.

5.2.6 `Vault.maxRedeemInternal` should always underestimate when user has a controller enabled

Severity: Medium Risk

Context: Vault.sol#L232-L238

Description: In the current implementation of `Vault.maxRedeemInternal`, the function underestimates the amount that the user can redeem/withdraw to **zero** if the `owner` has enabled the asset as collateral and has a controller enabled.

A more correct underestimation would be return `Shares.wrap(0)` when a controller has been enabled without checking if the asset has been enabled as collateral.

The current implementation of the `checkAccountStatus` of a Controller Vault inside the `RiskManager` contract will revert if the user is unhealthy, no matter what the operation was or if it involved the transfer or withdraw/redeem of a non-collateral (for the user) asset.

Recommendation: Consider increasing the underestimation performed by the `Vault.maxRedeemInternal` by checking only if a controller has been enabled for the `owner`. The Natspec documentation for `maxRedeem` and `maxWithdraw` should be updated accordingly.

Euler: Fixed as recommended in PR 163.

Spearbit: The PR mitigates the issue

5.2.7 `initVaultCache` can revert breaking liveness assumptions

Severity: Medium Risk

Context: Cache.sol#L80-L88, Cache.sol#L91-L92

Description: While some parts of the `initVaultCache` gracefully handle overflows, other parts can still revert:

```
// multiplication can overflow
uint256 newTotalBorrows =
                vaultCache.totalBorrows.toUint() * newInterestAccumulator /
↪  vaultCache.interestAccumulator;

// if newTotalBorrows didn't overflow, this shouldn't overflow either because it was divided by
↪  interestAccumulator > interestFee. (unless we use FullMath to compute newTotalBorrows)
uint256 feeAssets = (newTotalBorrows - vaultCache.totalBorrows.toUint()) * interestFee.toUint16()
                / (1e4 << INTERNAL_DEBT_PRECISION_SHIFT);
```

The guarantee described in the Whitepaper is broken:

> In the event that a vault encounters an overflow (either in rpow or its accumulator) the accumulator will stop growing, meaning that no further interest will be earned/charged. However, debts can still be repaid and funds withdrawn.

Recommendation: The `initVaultCache` function should not lock up as it would break liveness for important functions like `withdraw`, `redeem`, or `liquidate` that should always be possible to execute.

Euler: Fixed in PR 184.

Spearbit: The PR mitigates the finding. Spearbit suggests the following changes to be applied:

1) Write an inline comment that explains the reasoning behind the logic calculating `newInterestAccumulator` and `newTotalBorrows`. It would help a lot both Euler's new developers but also external SR that will look at the code to understand the codebase or find issues during bug bounties.

2) Update the EVK white paper, introducing a chapter about liveliness or expanding the existing one about "Interest Overflows".

Euler: Acknowledged. The white paper was updated to better cover this and similar effects. The Interest Overflows section should be an obvious source of information for anyone interested in this code.

5.3 Low Risk

5.3.1 Token defaults to `decimals == 18` when `decimals` reverts

Severity: Low Risk

Context: Token.sol#L32

When the `asset` contract obtained from `ProxyUtils.metadata` is a contract that does not implement the `decimals` function, the `staticcall` function will fail and return `18` as the asset's decimal.

This behavior could lead to a wrong assumption that could be dangerous if used in a price conversion. In addition to this problem, `decimals` have the following edge cases that should be considered:

- EOAs do not revert and will return `18`.
- Non-deployed contracts (`asset.code.length == 0`) do not revert and will also return `18`.
- Contracts that do not implement `decimals` do not revert and will also return `18`.
- Contracts that implement `decimals` but with a different return type (let's say `uint256`) do not revert if the returned value is `<= type(uint8).max`, and will hence return the value.

Recommendation: Euler should document all these edge cases and ensure that the `asset` received from `ProxyUtils.metadata()` at `Token.decimals()` is indeed a valid ERC20 compliant contract.

Euler: We acknowledge the issue. The vault makes sure the asset provided in the metadata is a contract in `initialize` function, where a check for non zero code length is performed. It handles a case when a low level call to an address without a code does not revert. In general though, it is not possible to ensure on-chain that any given contract is a valid and compliant implementation of ERC20.

Spearbit: Acknowledged.

5.3.2 Setting LTV configs without a configured oracle makes `EVK` unusable

Severity: Low Risk

Context: Governance.sol#L206-L229

Description: The `setLTV` function is used to set a new LTV config for a new or an existing collateral address.

However, it isn't checked whether the added collateral has a configured oracle. If collaterals are added without a configured oracle, most of the `EVK` functionality will be bricked. The `setLTV` function:

```
/// @inheritdoc IGovernance
function setLTV(address collateral, uint16 ltv, uint32 rampDuration) public virtual nonReentrant
↪   governorOnly {
    // self-collateralization is not allowed
    if (collateral == address(this)) revert E_InvalidLTVAsset();

    ConfigAmount newLTVAmount = ltv.toConfigAmount();
    LTVConfig memory origLTV = vaultStorage.ltvLookup[collateral];

    // If new LTV is higher than the previous, or the same, it should take effect immediately
    if (newLTVAmount >= origLTV.getLTV(true) && rampDuration > 0) revert E_LTVRamp();

    LTVConfig memory newLTV = origLTV.setLTV(newLTVAmount, rampDuration);

    vaultStorage.ltvLookup[collateral] = newLTV;

    if (!origLTV.initialized) vaultStorage.ltvList.push(collateral);

    emit GovSetLTV(
        collateral,
        newLTV.targetTimestamp,
        newLTV.targetLTV.toUint16(),
        newLTV.rampDuration,
        newLTV.originalLTV.toUint16()
    );
}
```

Recommendation: It's recommended to revert the execution of the `setLTV` function if the `oracle.getQuote(10 ** decimals, address(vaultCache.asset), collateral)` reverts or returns a value of `0`. Alternatively, the result of the `oracle.getConfiguredOracle` can be checked not to be `0`, as this indicates that no oracle is configured for this pair.

Euler: Acknowledged, no fix. The governor is considered trusted, and misconfigurations can't be prevented. Prefer to keep as is for simplicity.

Spearbit: Acknowledged.

5.3.3 Suppliers will be able to mint new shares even if `vaultCache.totalShares` is virtually above the `MAX_SANE_AMOUNT`

Severity: Low Risk

Context: Cache.sol#L98-L102, Cache.sol#L111-L114

Description: Like for the `totalBorrows`, the `Cache.initVaultCache` could also overflow for the `totalShares`. This scenario happens when part of the accrued interest must be accounted to the protocol/vault owner as fees (in shares). This is the logic that calculates the new total shares amount given `feeAssets > 0`:

```
if (feeAssets != 0) {
    uint256 newTotalAssets = vaultCache.cash.toUint() + OwedLib.toAssetsUpUint256(newTotalBorrows);
    newTotalShares = newTotalAssets * newTotalShares / (newTotalAssets - feeAssets);
    newAccumulatedFees += newTotalShares - vaultCache.totalShares.toUint();
}
```

The `newTotalShares` re-calculated to account for the fees could be virtually above the upper limit of `MAX_SANE_AMOUNT`. If we are in such a scenario, the function won't update `vaultCache.accumulatedFees` and `vaultCache.totalShares`:

```
if (newTotalShares != vaultCache.totalShares.toUint() && newTotalShares <= MAX_SANE_AMOUNT) {
    vaultCache.accumulatedFees = newAccumulatedFees.toShares();
    vaultCache.totalShares = newTotalShares.toShares();
}
```

As a consequence, all the accrued interest, until the new share amount is not overflowing anymore, will be accounted to the in total to the suppliers and not to the protocol/vault owner. Unlike the overflowing of the `totalBorrows` the not accounted `accumulatedFees` are lost forever for the protocol.

The second side effect of not accounting the shares to be minted to the protocol/vault owner is that users will be anyway able to mint new shares up to the delta `MAX_SANE_AMOUNT - vaultCache.totalShares` even if virtually, the real value of `vaultCache.totalShares` would be already above `MAX_SANE_AMOUNT`.

Recommendation: Euler should document the side effects of this scenario in the "Interest Overflows" of the EVK white paper.

Euler: We acknowledge the issue, no fix in code, white paper updated to better cover this and similar effects.

We accept that at numerical limits accounting will break with many side effects. We only intend to make sure it is still possible to interact with the vault (it's not bricked).

Spearbit: Acknowledged.

5.3.4 `Token` transfer methods should not allow `from == address(0)` and `to == address(0)`

Severity: Low Risk

Context: Token.sol#L51-L53, Token.sol#L56-L58, Token.sol#L61-L73

Description: The functions `transfer`, `transferFrom` and `transferFromMax` allow the caller to specify arbitrary `from` and `to`. Both the input parameters are allowed to assume the `address(0)` value.

Both `transfer` and `transferFromMax` internally will execute `transferFrom` with some custom logic depending on which function is executed.

```
/// @inheritdoc IERC20
function transfer(address to, uint256 amount) public virtual reentrantOK returns (bool) {
    return transferFrom(address(0), to, amount);
}

/// @inheritdoc IToken
function transferFromMax(address from, address to) public virtual reentrantOK returns (bool) {
    return transferFrom(from, to, vaultStorage.users[from].getBalance().toUint());
}

/// @inheritdoc IERC20
function transferFrom(address from, address to, uint256 amount) public virtual nonReentrant returns
↪   (bool) {
    (, address account) = initOperation(OP_TRANSFER, from == address(0) ? CHECKACCOUNT_CALLER : from);

    if (from == address(0)) from = account;
    if (from == to) revert E_SelfTransfer();

    Shares shares = amount.toShares();

    decreaseAllowance(from, account, shares);
    transferBalance(from, to, shares);

    return true;
}
```

Allowing to transfer shares to the address(0) (to = address(0)) should not be permitted, given that the same behavior in BalanceUtils.increaseBalance will result in a revert.

Allowing to execute transferFromMax with from = address(0) (with the ability to perform the transfer to address(0)) will instead enable a funky behavior. In this case, the balance of address(0)will be used as theamountbuttransferFromwill use themsg.sender‘ as the caller.

Let's see an example:

1) Alice owns 3e18 shares.
2) Alice calls transfer(address(0), 1e18) sending 1e18 shares to address(0).
3) Alice calls transferFromMax(address(0), bob), she wants to transfer her whole balance of 2e18 shares to Bob.

When transferFromMax is executed, Alice owns 2e18 shares, but the function will use the balance of address(0) as the source of the amount to be transferred (vaultStorage.users[from].getBalance().toUint()). Inside transferFrom, the from value will be changed from address(0) to Alice and will transfer 1e18 (the shares accounted in the balance of address(0)) from Alice to Bob.

The result is that Alice has not transferred her whole balance of 2e18 to Bob but just 1e18.

Recommendation: Euler should:

- Revert inside all the functions when to == address(0) (this logic can be moved inside _transferFrom, see below).
- Revert in transferFromMax and transferFrom when from == address(0).
- Refactor the transferFrom function to still support the transfer special case where from is forced as address(0).
  1) Create a _transferFrom private function where all the code is moved to.
  2) transfer, transferFrom and transferFromMax will call _transferFrom.
  3) _transferFrom will revert when to == address(0).
  4) transferFrom and transferFromMax reverts if from == address(0).

Euler: Fixed in PR 182.

Spearbit: The provided PR mitigates the issue

5.3.5 Borrowers could be able to borrow avoiding the borrowCap

Severity: Low Risk

Context: RiskManager.sol#L94-L102

Description: Let's assume that the borrowCap has been configured with a value near MAX_SANE_DEBT_AMOUNT and that we are in a situation where Cache.initVaultCache has overflown when the snapshot has been taken.

Cache.initVaultCache overflows when totalBorrows + the accrued interest would be bigger than MAX_SANE_DEBT_AMOUNT. In this case, both totalBorrows and interestAccumulator are not updated in the vault cache and storage.

Let's also assume that when the snapshot was taken totalBorrows (that because of the overflow does not include the accrued interest) is below vaultCache.borrowCap even if in theory it would be virtually already above such cap.

Given these premises, a borrower could be able to perform a borrowing operation avoiding the borrow caps if the amount borrowed is lower than totalBorrows - MAX_SANE_DEBT_AMOUNT.

When if (borrows > vaultCache.borrowCap && borrows > prevBorrows) revert E_BorrowCapExceeded(); is evaluated, snapshot.borrows has been initialized with the cached version of the totalBorrows that was not including the accrued interest.

Recommendation: Euler should consider storing in the cache the information related to the overflow status and use it in addition to the current logic to evaluate if the borrowing operation should be allowed or not.

Euler: We acknowledge the issue, no fix in code. The white paper was updated to better cover this and similar effects.

We accept that at numerical limits accounting will break with many side-effects. We only intend to make sure it is still possible to interact with the vault (it's not bricked).

Spearbit: Acknowledged.

5.3.6 Borrowers will be able to borrow even if the `totalBorrows` is virtually already above the `MAX_SANE_DEBT_AMOUNT`

Severity: Low Risk

Context: Cache.sol#L106-L115

Description: Let's assume that there's no borrow cap configured or that the borrow cap is very near the `MAX_SANE_DEBT_AMOUNT`. Let's also assume that the current `totalBorrows` + accrued interest overflows the `MAX_SANE_DEBT_AMOUNT` value.

In this scenario, the `Cache.initVaultCache` will not accrue the interest into `totalBorrows` to avoid overflowing and the `totalBorrows` and `interestAccumulator` value will remain unchanged in both the vault cache and storage.

In this case, a borrower could be able to perform a borrow operation if the amount borrowed is less than the delta `MAX_SANE_DEBT_AMOUNT - totalBorrows`. This behavior should be forbidden given that:

- The `totalBorrows` is virtually already over the `MAX_SANE_DEBT_AMOUNT` is we consider the accrued interest.
- The borrower was able to open a borrowing position with a favorable non-updated `interestAccumulator`.

Recommendation: Euler should consider storing in cache the information related to the overflow status and revert any operation that would increase the borrowing amount.

Euler: We acknowledge the issue, no fix in code. The white paper was updated to better cover this and similar effects.

We accept that at numerical limits accounting will break with many side-effects. We only intend to make sure it is still possible to interact with the vault (it's not bricked).

Spearbit: Acknowledged.

5.3.7 `setHookConfig` and `setConfigFlags` should validate the new flags value

Severity: Low Risk

Context: Governance.sol#L257-L266, Governance.sol#L268-L272

Description: Both `Governance.setHookConfig` and `Governance.setConfigFlags` functions allow the caller to set an arbitrary value of the flags without performing any sanity checks. This means that the user could enable flags that are not currently supported by the current implementation of the vault.

If future implementation of the `EVK` will use those flags, the vault instance could act in an unexpected way (reverts, can't withdraw, redeem, borrow or in general is disrupted). The scenario would be even more problematic if the vault has also renounced to the ownership and the flags cannot be changed anymore.

Recommendation: Euler should perform sanity checks on the flags value passed to `Governance.setHookConfig` and `Governance.setConfigFlags` and revert if they are outside the range of values supported by the current implementation of `EVK`.

Euler: Fixed as recommended in PR 165.

Spearbit: The provided PR mitigates the issue.

5.3.8 `Governance` should not be able to change the vault's `name` and `symbol`

Severity: Low Risk

Context: Governance.sol#L181-L191

Description: The current implementation of the `Governance` contract allows the governor to update at any point and with any value, even an empty one, both the `name` and `symbol` of the deployed `EVault`. These values are later on used in `Token.name` and `Token.symbol`.

Allowing such behavior could create confusion and could be leveraged by malicious users to pursue attack vectors like scams or code injections.

Recommendation: Euler should:

- Not allow the governor to update the `name` and `symbol` state values once they have been initialized.
- Not allow the `name` and `symbol` to assume the empty value.

A better solution that improves both the security and gas consumption would be to have both `name` and `symbol` as `immutable` values that can be initialized only during the deployment phase.

Euler: Fixed in PR 64. Use of immutable strings would be difficult with current Solidity support for immutables. With current trailing data design, sending name and symbol in every proxy call would be wasteful in terms of gas.

Spearbit: The PR mitigates the issue, but Spearbit has some suggestions:

1) Unless there's a proper reason, Euler should revert when `getTokenSymbol` low-level `staticcall` return `success == false`. This will allow the creation of confusing `EVault` with symbols like `e-UNDEFINED-1`, `e-UNDEFINED-2` and so on. From the user prospective, these vaults could be seen as scam vaults or at best non-functional vaults.
2) Euler should revert if the value of `underlyingSymbol` is empty. This would create an `EVault` with `vaultStorage.symbol = e-1` and `vaultStorage.name = EVK Vault e-1`
3) Gas Optimization: save the vault's symbol in a local variable and use the local variable to initialize `vaultStorage.symbol` and `vaultStorage.name`. This will avoid the second `SLOAD` of `vaultStorage.symbol`

Euler: Acknowledged. While we agree all foreseeable assets, which will be used to create the vaults, should have a proper `symbol()` implementations, we chose not to create a hard dependency on this particular view function, especially that we also handle failing `decimals()` call, which is a result of an audit recommendation. We don't believe it to be a security threat, and it makes the vault implementation as generic as possible. As for gas optimizations, we don't consider it a concern in the `initialize` function and prefer to keep the code simple.

5.3.9 Consider reverting the flashloan operation if the returned amount of is not exactly the original balance

Severity: Low Risk

Context: Borrowing.sol#L176

Description: The current `flashloan` logic will revert if the new `EVault` balance is lower compared to the one snapshotted before the `flashloan`.

```
if (asset.balanceOf(address(this)) < origBalance) revert E_FlashLoanNotRepaid();
```

With such logic, the `flashloan` function allows, without any valid reason, the caller to transfer more `asset` than required. In such a scenario, the user will be forced to include in the batch a `skim` execution, otherwise, the surplus "donated" to the vault will be lost (skimmed by someone else in the future).

Recommendation: Euler should consider reverting the `flashloan` function if the user has not returned the exact amount expected.

```
- if (asset.balanceOf(address(this)) < origBalance) revert E_FlashLoanNotRepaid();
+ if (asset.balanceOf(address(this)) != origBalance) revert E_FlashLoanNotRepaid();
```

Euler: Acknowledged, no fix.

There are legitimate use cases for leaving more tokens in the vault after repaying the flash-loan e.g.:

- There is some dust excess after repay and the user might want to get rid of it for a gas refund.
- The flashloan might be a part of a larger batch, and the excess will be skimmed in subsequent operation.

Spearbit: Acknowledged.

5.3.10 `CFG_EVC_COMPATIBLE_ASSET` should be `immutable` and not be allowed to be changed

Severity: Low Risk

Context: AssetTransfers.sol#L28, Governance.sol#L268-L272

Description: The role of the governance config flag `CFG_EVC_COMPATIBLE_ASSET` is to ensure that the underlying vault asset is not transferred to a subaccount (in the EVC context) if such asset is not EVC compatible.

The name of the flag, its meaning, and its role are self-explanatory and very explicit. Such a flag should be set to `true` when the underlying asset of the `EVault` is an EVC-compatible asset, and to `false` if otherwise it's a "normal" ERC20-like token.

Currently, the `Governance` module allows the owner to change the value of such flag at any moment and to a value that could be wrong given the `EVault` configuration given that there is no validation between the flag's value and the underlying vault's asset.

Given such premises, we suggest to:

- Set the `CFG_EVC_COMPATIBLE_ASSET` flag as an immutable value.
- Initialize the `CFG_EVC_COMPATIBLE_ASSET` flag when the `EVault` is initialized.
- Initialize the flag to `true` if the `asset` exposes the `EVC()` getter and if the address returned by such getter is equal to the `evc` address used for the `EVault` just deployed.

Recommendation: Euler should consider implementing the recommendations listed above.

Euler: Acknowledged, no fix.

From the vault's perspective it's impossible to objectively verify if an asset is EVC compatible. Therefor we prefer to leave this decision to the creator of the vault, which could also be a smart-contract which calls `EVC()` getter.

As for storing the flag as a proxy metadata, we consider the metadata to be suitable for data that is useful in most calls and saves storage reads. Neither condition is met by the flag - it's only relevant when assets are pushed and the flag is already available in a warm slot in such cases.

Spearbit: Acknowledged.

5.3.11 Virtual shares steal interest

Severity: Low Risk

Context: Shares.sol#L21-L26

Description: When redeeming (or withdrawing) vault shares, the conversion uses the total shares including the virtual shares to compute the principal and interest earned:

```
// in `redeem`
// shares * (totalAssets + 1e6) / (totalShares + 1e6)
Assets assets = shares.toAssetsDown(vaultCache);

function toAssetsDown(Shares amount, VaultCache memory vaultCache) internal pure returns (Assets) {
    (uint256 totalAssets, uint256 totalShares) = ConversionHelpers.conversionTotals(vaultCache);
    unchecked {
        return TypesLib.toAssets(amount.toUint() * totalAssets / totalShares);
    }
}

library ConversionHelpers {
    // virtual deposit used in conversions between shares and assets, serving as exchange rate
↪   manipulation mitigation
    uint256 constant VIRTUAL_DEPOSIT_AMOUNT = 1e6;

    function conversionTotals(VaultCache memory vaultCache)
        internal
        pure
        returns (uint256 totalAssets, uint256 totalShares)
    {
        unchecked {
            totalAssets =
                vaultCache.cash.toUint() + vaultCache.totalBorrows.toAssetsUp().toUint() +
↪   VIRTUAL_DEPOSIT_AMOUNT;
            totalShares = vaultCache.totalShares.toUint() + VIRTUAL_DEPOSIT_AMOUNT;
        }
    }
}
```

Therefore, the virtual shares have their own fair share on the total assets (including virtual assets), essentially earning interest and locking it up. This interest cannot be withdrawn as the virtual shares are not owned by anyone.

Recommendation: As the interest earned by the virtual shares will be negligible for most vaults and fixing the issue will likely introduce other conversion and price share issues, we recommend accepting that a small part of real interest is locked up.

Euler: We acknowledge and agree with the recommendation.

Spearbit: Acknowledged.

5.3.12 Immutable `EVK` vault creation via `GenericFactory` could be frontrunned by an update of the `implementation`, different from the one chosen by the caller

Severity: Low Risk

Context: GenericFactory.sol#L78-L99

Description: `GenericFactory.createProxy` is the function that anyone should use to deploy a valid and recognized `EVK` compatible vault. The function allows the caller to specify an `upgradeable` parameter that when it's `false` will deploy an immutable vault using the current value of the state variable `implementation`.

This state variable can be changed at any time by the `GenericFactory` admin via the `setImplementation` function. The EVK white paper states that:

> After creating an immutable vault, the vault's implementation should be confirmed to be the desired version, since it could've been changed by the factory admin prior to vault creation.

It's important to allow the vault creator to ensure that the vault will be created with the desired and expected `implementation` without the risk of being frontrunned on purpose or mistakenly.

Recommendation: Euler should allow the vault creator to specify the desired `implementation` inside the `createProxy` function parameters. If the value differs from the one inside the `implementation` state variable, the transaction should revert.

Euler: Fixed in PR 195.

Spearbit: The provided PR mitigates the issue.

5.3.13 `IRMSynth` should revert when deployed with a non-compatible `oracle`

Severity: Low Risk

Context: IRMSynth.sol#L37

Description: Unlike the `IRMLinearKink` which does not have any external dependencies, the `IRMSynth` IRM has 3 different dependencies.

Given that the oracle is the main dependency, the deployment of the IRM should revert if the very first call reverts or returns an invalid value. This means that the oracle has one of the following problems:

- It's not an Euler oracle.
- It has not been correctly configured to support `synth` and `referenceAsset`.
- It's not working as expected, given that the returned price is 0.

Recommendation: Euler should revert the deployment of `IRMSynth` if `oracle.getQuote(1e18, synth, referenceAsset)` returns an unexpected value like `0`.

Euler: Fixed in commit 67a74dfd.

Spearbit: The provided commit 67a74dfd mitigates the issue.

5.3.14 `EulerSavingRate` `gulp` can delay the full accrual of the user's interest

Severity: Low Risk

Context: EulerSavingsRate.sol#L134-L148

Description: `gulp` is the ESR mechanism that starts the accrual of the amount of asset that has been sent to the ESR by an external entity. Once gulp if called, such amount is added to `esrSlot.interestLeft` and `esrSlot.interestSmearEnd` is resetted to `block.timestamp + INTEREST_SMEAR`

Because `gulp` can be called by anyone, at any time and without any restriction on the amount of interest to be added to `interestLeft`, such a mechanism could be abused by attackers that could delay the full accrual of the interest of existing users.

Let's assume that there's no user in the `ESR`:

1) Alice deposits `10e18`.
2) `10e18` interests are added to the `ESR`.
3) Alice calls immediately `gulp` to start the accrual of those interests. Alice would assume that after 2 weeks she should have accrued all those interests and be able to withdraw `~20e18` of the underlying asset with her shares.
4) When `esr.INTEREST_SMEAR() / 2` has passed, someone calls `gulp()` that would postpone of another `esr.INTEREST_SMEAR()` seconds the full accrual of the interest. Because `gulp` calls `updateInterestAndReturnESRSlotCache` and because half of the time has passed, half of the `10e18` interest are added to `interestLeft`.
5) Alice waits another `esr.INTEREST_SMEAR() / 2` seconds and calls `esr.withdraw(max, ...)` expecting to get `20e18` of assets, but only half of the new `interestSmearEnd` has passed so she has matured only half of the remaining `interestLeft` that is `~2.5e18`.

Alice has withdrawn only ~17499999999999999999 instead of the full 20e18 she should have got if no one would have gulped meanwhile.

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "./lib/ESRTest.sol";

contract QuickESRTest is ESRTest {

    function testMultipleGulps() public {
        address alice = makeAddr("esrAlice");
        uint256 depositAmount = 10e18;
        doDeposit(alice, depositAmount);
        assertEq(asset.balanceOf(alice), 0);

        // 10e18 of interest are deposited but no one gulp them
        asset.mint(address(esr), 10e18);
        esr.gulp();

        skip(esr.INTEREST_SMEAR() / 2);
        esr.gulp();

        skip(esr.INTEREST_SMEAR() / 2);
        uint256 maxW = esr.maxWithdraw(alice);
        uint256 balance = esr.balanceOf(alice);

        console.log("maxW", maxW);
        console.log("balance", balance);

        maxW = esr.maxWithdraw(alice);
        vm.prank(alice);
        esr.withdraw(maxW, alice, alice);

        console.log("asset balance", asset.balanceOf(alice));
    }
}
```

Euler: We acknowledge the issue, no fix.

Since interest is expected to constantly flow into the savings rate the issue of in perpetuity delaying interest payments is non existent as the flow will still be somewhat constant. There's no risk of principal loss for users, neither is there an economic incentive for anyone to grief users in this manner.

Even in the event of a one-time gulp, yes a portion the smear can be delayed indefinitely, but the affected amount decreases exponentially over time.

Since in practice this issue is non existent, not economically viable and there's no risk of user funds getting lost I disagree with the severity label.

Will keep as is.

Spearbit: Acknowledged.

5.3.15 `ESynth` should only allow the execution of `allocate` and `deallocate` to and from EVC-compatible vault

Severity: Low Risk

Context: ESynth.sol#L81-L96

Description: The `allocate` and `deallocate` functions allow the `ESynth` contract to deposit and withdraw `ESynth` tokens from EVC compatible vaults specified by the caller in the function input parameter. The specified `vault` should follow these requirements:

- It's an EVC-compatible vault.
- Use the same `EVC` address used by the `ESynth`.

Both these requirements are not checked when those functions are executed.

Furthermore, when an allocate is performed, the vault is added to the ignore total supply. If the deallocation is happening for the full allocated amount, the vault is not deleted from the ignored total supply.

Recommendation: Euler should revert the transaction if the specified `vault` does not implement the `EVC()` getter or if the addresses returned by the getter are different from the value stored in the inherited `EVCUtil.evc` state variable.

For the second issue, consider documenting the fact that deallocation does not guarantee the auto-removal from the total ignored supply. Another idea would be to track the allocated/deallocated amounts and if it hits 0, to auto-remove the vault.

Euler: For the second issue, we're aware of this. We did it the way it is because we didn't want to overcomplicate the code and remove from ignoredForTotalSupply when 0 is hit on deallocation. Even if that's the case, apart from totalSupply being slightly more expensive to call, there's no side effects. The synths will be actively managed and the synth owner can always remove the address from the list if needed.

Spearbit: The provided commit d589304 mitigates the first issue.

5.3.16 `IRMSynth` special-cases oracle price of `0`

Severity: Low Risk

Context: IRMSynth.sol#L69-L72

Description: The `IRMSynth._computeRate` code treats a quoted amount of `0` as an error condition. However, the Euler oracles revert on error conditions and don't use a return value of `0` to indicate errors.

Recommendation: A quoted amount of `0` should be treated as any other tiny quoted amount, meaning the rate should be adjusted.

Euler: Fixed in commit 3c1fb588.

Spearbit: The provided commit mitigates the issue.

5.3.17 Non-standard ERC20 behavior for `EVault` with `from=address(0)`

Severity: Low Risk

Context: Token.sol#L64

Description: The `Token` module that is used for the `eVault` ERC20 shares handles a `transferFrom` (and `transferFromMax`) with `from=address(0)` as a transfer from the EVC-authenticated account (usually `msg.sender`). When integrating eVault ERC20 tokens this special behavior is unexpected and could lead to incompatibilities. Note that this is still an issue even if the integrator does not use the EVC as the entrypoint and just treats the vault as a standard ERC20 token.

Recommendation: Consider removing this special case or document it.

Euler: The transfer logic has been refactored to remove the behaviour, in PR 182.

Spearbit: The provided PR mitigates the issue.

5.3.18 Fee shares are minted at worse price for fee receivers

Severity: Low Risk

Context: Cache.sol#L95-L102

Description: Whenever interest is accrued, part of the interest is taken as a fee. This fee is used to mint shares for the "fee receivers". The code should work like adding `(newInterest - feeAssets)` to `totalAssets`, then depositing `feeAssets` into the vault to mint new shares for them. However, it is currently ignoring the `VIRTUAL_DEPOSIT_AMOUNT = 1e6` of the `deposit` step conversion. This leads to the fee receivers receiving fewer shares in practice, compared to them receiving the fee as "assets" and depositing themselves. (As they mint at a higher share price `totalAssets / totalShares > (totalAssets + VIRTUAL_DEPOSIT_AMOUNT) / (totalShares + VIRTUAL_DEPOSIT_AMOUNT)` for most vaults.)

Recommendation: The share price difference gets less relevant the smaller the virtual amounts are compared to `totalAssets` and `totalShares`. For non-pathological vaults with some deposited assets and some borrowing behavior, the difference becomes negligible. Still, consider first updating the `vaultCache.totalBorrows` by `newInterest - feeAssets` followed by computing `Shares newShares = feeAssets.toSharesDown(vaultCache);`, and increasing the `totalBorrows` by the `feeAssets`, and `totalShares` by `newShares`.

Spearbit: Euler decided to not implement the recommendations provided but to document the behavior in PR 182.

5.3.19 `ESynth.mint` can emit 0-`Transfer` events for non-allowed minters

Severity: Low Risk

Context: ESynth.sol#L43, ESynth.sol#L63

Description: The mint function can currently be called by anyone, even if they're not an authorized minter. Authorized minters are the ones for whom the admin has set capacity.

In order to avoid this, it should revert if `amount == 0`, what would prevent the executions and emitting `Transfer(address(0), account, 0)` event when `_mint(...)` is called. Mint checks involving `capacity` should add logic in order to handle cases where `capacity == 0 && amount == 0`, as it will return false allowing the execution of `mint` function.

Same allowed behavior happens on `burn` function when using `amount == 0` even if you shouldn't be able to effectively burn from an account.

Recommendation: Add checks to avoid non-allowed minters to mint.

Euler: Fixed in commits 090cc37b and 651a2fe0. Chose to return early when `amount == 0` to prevent unexpected behavior for integrations which mint 0 on accident, and also implemented an early return on burn.

Spearbit: The issue has been mitigated by the commits 090cc37b and 651a2fe0.

5.3.20 `evc` used by the `PegStabilityModule` and `ESynth` contract may not be the same

Severity: Low Risk

Context: PegStabilityModule.sol#L35

Description: `PegStabilityModule` constructor stores an `ESynth` address which is not enforced to be related to the corresponding `EVC` used. Add a sanity check that enforces that `PegStabilityModule` `EVC` is the same one used by the `ESynth` contract. To be able to do that, the `evc` variable inside `EVCUtil` should be exposed. Now it's an internal one without any getter.

Recommendation: Make the following change in `EVCUtil.sol`:

```
  abstract contract EVCUtil {
-     IEVC internal immutable evc;
+     IEVC public immutable evc;
```

And verify at PegStabilityModule constructor

```
if (_synth.evc != evc) revert ErrorDifferentEVC();
```

Euler: Acknowledged. Will keep as is. Each network will have a single EVC instance so I don't foresee any issues.

Spearbit: Acknowledged.

5.3.21 PegStabilityModule **should change sanity checks from** > **to** >=

Severity: Low Risk

Context: PegStabilityModule.sol#L27

Description: toUnderlyingFeeBPS and toSynthFeeBPS must be < BPS_SCALE otherwise quoteToUnderlyingGivenOut and quoteToSynthGivenOut will revert because of division by zero.

Also, when they are equal to BPS_SCALE the user would get nothing back when they swap amountIn of ESynth/underlying for underlying/ESynth.

Recommendation: Change the sanity checks from > to >=.

```
- if (toUnderlyingFeeBPS > BPS_SCALE || toSynthFeeBPS > BPS_SCALE) {
+ if (toUnderlyingFeeBPS >= BPS_SCALE || toSynthFeeBPS >= BPS_SCALE) {
      revert E_FeeExceedsBPS();
  }
```

Euler: Fixed in commit 43c93c6d.

Spearbit: The issue has been mitigated by commit 43c93c6d.

5.3.22 PegStabilityModule quoteToSynthGivenOut **should round in favor of the protocol**

Severity: Low Risk

Context: PegStabilityModule.sol#L89

Description: The value returned by quoteToSynthGivenOut:

```
function quoteToSynthGivenOut(uint256 amountOut) public view returns (uint256) {
    return amountOut * BPS_SCALE / (BPS_SCALE - TO_SYNTH_FEE);
}
```

Will be used to quote how many underlying tokens the user must pay to get back amountOut of ESynth tokens when called at swapToSynthGivenOut:

```
function swapToSynthGivenOut(uint256 amountOut, address receiver) external returns (uint256) {
    uint256 amountIn = quoteToSynthGivenOut(amountOut);

    underlying.safeTransferFrom(_msgSender(), address(this), amountIn);
    synth.mint(receiver, amountOut);

    return amountIn;
}
```

Recommendation: This amount must be rounded up to favor the protocol and not the user.

Euler: Acknowledged. Since there is already a fee being paid any tiny rounding inaccuracy will be insignificant and does not warrant the added complexity to round it. Will keep as is.

Spearbit: Acknowledged.

5.3.23 `PegStabilityModule` `quoteToUnderlyingGivenOut` should round in favor of the protocol

Severity: Low Risk

Context: PegStabilityModule.sol#L81

Description: The value returned by `quoteToUnderlyingGivenOut`:

```
function quoteToUnderlyingGivenOut(uint256 amountOut) public view returns (uint256) {
    return amountOut * BPS_SCALE / (BPS_SCALE - TO_UNDERLYING_FEE);
}
```

Will be used to determine the amount of `Esynth` assets that the user needs to pay to get back `amountOut` of underlying assets when calling `swapToUnderlyingGivenOut`:

```
function swapToUnderlyingGivenOut(uint256 amountOut, address receiver) external returns (uint256) {
    uint256 amountIn = quoteToUnderlyingGivenOut(amountOut);

    synth.burn(_msgSender(), amountIn);
    underlying.safeTransfer(receiver, amountOut);

    return amountIn;
}
```

This value should favor the protocol and not the user.

Recommendation: The value should be **rounded up** and not rounded down like now.

Euler: Acknowledged. Since there is already a fee being paid any tiny rounding inaccuracy will be insignificant and does not warrant the added complexity to round it. Will keep as is.

Spearbit: Acknowledged.

5.3.24 Single-step admin transfer can be risky

Severity: Low Risk

Context: GenericFactory.sol#L111, ESynth.sol#L5

Description: `GenericFactory.sol` implements the role of `upgradeAdmin` which performs the action of setting new `implementation`s or setting a new `upgradeAdmin`. It uses a single-step role transfer design, which adds the risk of setting an unwanted role owner by accident. If the ownership transfer is not done with excessive care it can be lost forever.

Similarly, it happens with the Open Zeppelin `Ownable` library at `ESynth` contract, which could be `Ownable2Step` to avoid possible problems.

Recommendation: Consider using a two-step ownership transfer mechanism for critical admin changes, which would avoid typos and "*fat finger*" mistakes.

Some good implementations of the two-step ownership transfer pattern can be found at Open Zeppelin's `Ownable2Step` or Synthetic's `Owned`.

Euler: Acknowledged, no fix. While we understand the proposed solution improves the security of privileges transfer, we consider them not significant enough compared to increased code complexity.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 `delegateToModuleView`'s `caller` encoding can be packed

Severity: Gas Optimization

Context: Dispatch.sol#L122, ProxyUtils.sol#L22

Description: The `delegateToModuleView` function appends the `caller` address for other `view` functions that are delegatecall'd into. It can be read using `ProxyUtils.useViewCaller()`. It's currently appended as a 32-bytes value (with the upper 12 bytes being zero).

Recommendation: Consider optimizing the calldata by appending only the 20-bytes of the address, removing the 12 leading zero bytes:

```
mstore(0, 0x1fe8b95300000000000000000000000000000000000000000000000000000000)
let strippedCalldataSize := sub(calldatasize(), PROXY_METADATA_LENGTH)
// we do the mstore first offset by -12 so the 20 address bytes align right behind 36 +
↪  strippedCalldataSize
// note that it can write into the module address if the calldata is less than 12 bytes, therefore
↪  write before we write module
mstore(add(24, strippedCalldataSize), caller())
mstore(4, module)
calldatacopy(36, 0, strippedCalldataSize)
// insize: stripped calldatasize + 36 (signature and module address) + 20 (caller address)
let result := staticcall(gas(), address(), 0, add(strippedCalldataSize, 56), 0, 0)
```

Euler: Fixed in PR 215.

Spearbit: Verified.

5.4.2 Avoid using `+=`, `-=` operators for storage variables

Severity: Gas Optimization

Context: EulerSavingsRate.sol#L122, EulerSavingsRate.sol#L130, EulerSavingsRate.sol#L160

Description: `+=` and `-=` operations on storage variables are cheaper if declared as `totalAssetsDeposited = totalAssetsDeposited + assets`.

Gas optimization from this is ~15-30 gas per call/instance. Over 948 in the tests after the 3 instances change.

Recommendation: Consider avoiding `+=`, `-=` operators for storage variables as they are less gas efficient.

Euler: Fixed in commit 30cfa84f.

Spearbit: The issue has been mitigated by commit 30cfa84f.

5.4.3 `require`s using input parameters should go right after the function declaration

Severity: Gas Optimization

Context: BeaconProxy.sol#L26

Description: `require` statements of input parameters are commonly declared right after the function declaration to avoid executing extra logic in the case of an inevitable revert.

Recommendation: Move the `require` statement to the first line within the function.

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.4.4 Logic only used can be inlined in order to save gas

Severity: Gas Optimization

Context: LiquidityUtils.sol#L66-L67

Description: Some logic in `checkNoCollateral` is only used once and can be inlined in order to save gas and simplify the logic by reducing steps.

Recommendation: Consider the following change

```
- uint256 balance = IERC20(collateral).balanceOf(account);
- if (balance > 0) return false;
+ if (IERC20(collateral).balanceOf(account) != 0) return false;
```

Euler: Fixed in commit 430a0531.

Spearbit: Verified.

5.4.5 Immutable variables are more gas efficient than storage variables

Severity: Gas Optimization

Context: Core.sol#L15-L16

Description: The `governor` and `feeReceiver` addresses are declared as variables `public`. When only assigned once, variables should be marked as immutable for gas optimization, reducing the number of `SLOAD` operations and improving performance.

Recommendation: Set these variables to immutable for gas optimization.

Euler: Acknowledged. Contract was removed from scope.

Spearbit: Acknowledged.

5.4.6 Variable can be cached to save gas

Severity: Gas Optimization

Context: GenericFactory.sol#L79

Description: Multiple accesses to `implementation` in `createProxy` function leads to inefficiencies in gas usage. Caching it into a local variable can save gas and simplify the code.

Recommendation: Ensure `implementation` is cached into a local variable and consistently used throughout the function. For example:

```
function createProxy(bool upgradeable, bytes memory trailingData) external nonReentrant returns
↪   (address) {
    address _implementation = implementation;
    if (_implementation == address(0)) revert E_Implementation();
    // ...
```

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.4.7 Use custom errors for consistency and gas savings

Severity: Gas Optimization

Context: BeaconProxy.sol#L26

Description: Although the usage of custom errors is generalized in the repository, a `require` condition is used instead of a custom error in this case. This should be resolved to keep consistency and save some gas.

```
require(trailingData.length <= MAX_TRAILING_DATA_LENGTH, "trailing data too long");
```

Recommendation: Use a custom error instead:

```
- require(trailingData.length <= MAX_TRAILING_DATA_LENGTH, "trailing data too long");
+ if (trailingData.length > MAX_TRAILING_DATA_LENGTH) revert TrailingDataTooLongError();
```

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 `BorrowUtils.decreaseBorrow` behavior to round up debt should be documented

Severity: Informational

Context: BorrowUtils.sol#L57-L70

Description: Unlike `increaseBorrow` and `transferBorrow` that cast the `assets` amount (of type `Assets`) to `Owed` and interact with the user's debt position in `Owed` terms, the `decreaseBorrow` operation logic applies the inverse behavior, rounding up the user's exact debt, casting it to `Assets` and then interact with the repaid `amount` in `Assets` terms.

Then it proceeds to update both the user's debt balance and `totalBorrows` recasting the remaining amount of debt to `Owed`. Following this behavior, the remaining debt of the users (and the `totalBorrows` as a consequence) will be saved as a rounded up version.

Recommendation: Euler should document this behavior and ensure that the `decreaseBorrow` function cannot be called with `amount = 0` (like it's already enforced in the current codebase) to prevent increasing the user's effective debt in a decrease-debt operation.

Euler: Fixed in PR 198.

Spearbit: The issue has been mitigated by PR 198.

5.5.2 Violators can temporarily prevent liquidations by frontrunning the liquidation transaction and slightly increasing their position health

Severity: Informational

Context: Liquidation.sol#L103

Description: In a liquidation transaction, the liquidator specifies `repayAssets`, which represents the amount of underlying debt transferred from the violator to the sender. The liquidate function invokes the `calculateLiquidation` function to perform necessary liquidation calculations.

At the end of the `calculateLiquidation` function, there is a check ensuring that `desiredRepay` is less than or equal to `repay`, calculated based on the violator's liabilities and collateral balance.

```
function calculateLiquidation(
    VaultCache memory vaultCache,
    address liquidator,
    address violator,
    address collateral,
    uint256 desiredRepay
) private view returns (LiquidationCache memory liqCache) {
    // Init cache
    // . . .

    // Checks
    // . . .

    liqCache = calculateMaxLiquidation(liqCache, vaultCache);

    // Adjust for desired repay

    if (desiredRepay != type(uint256).max) {
        uint256 maxRepay = liqCache.repay.toUint();
        if (desiredRepay > maxRepay) revert E_ExcessiveRepayAmount(); // <---

        if (maxRepay > 0) {
            liqCache.yieldBalance = desiredRepay * liqCache.yieldBalance / maxRepay;
            liqCache.repay = desiredRepay.toAssets();
        }
    }
}
```

Violators can *slightly decrease* their `maxRepay` by either increasing their collateral or decreasing their borrowings, without bringing their positions back to health. This might cause the liquidation transaction to revert if the liquidator has performed a partial liquidation or specified an amount close to `maxRepay`.

Recommendation: Assess whether this behavior is desirable and consider documenting this scenario. Alternatively, rather than reverting when `desiredRepay > maxRepay`, the `EVK` could proceed with `maxRepay` as the repayment.

Euler: Acknowledged, no fix. We consider the scenario easily circumvented by bots performing liquidations through smart contracts or by setting `desiredRepay` to max uint256.

Spearbit: Acknowledged.

5.5.3 Potential reorg attack risk for `GenericFactory` deployments on `L2s`

Severity: Informational

Context: GenericFactory.sol#L84, MetaProxyDeployer.sol#L40

Description: The `GenericFactory` uses `CREATE` instead of `CREATE2` for deploying the `EVault` proxies. Theoretical reorgs on L2s could enable a malicious deployment to divert funds from a legitimate proxy by utilizing the deposits made to this proxy.

In a Slack conversation, it was mentioned that there are very loose plans to deploy the project on other chains.

An example attack scenario includes the following steps:

- `Alice` creates an `EVault` via the factory contract in `Transaction A`.
- Alice deposits into the `EVault` in `Transaction B`.
- A block reorg occurs, causing `Transaction A` to be discarded while `Transaction B` remains.
- Normally, `Transaction B` would revert if executed.
- `Bob` then deploys the `EVault` that `Alice` initially created, using the same address.

- The deposit made by Alice now goes to Bob's vault, performs some malicious actions by using the governor-only functions.

More information on Blockchain Reorgs can be found in the Blockchain reorgs for Managers and Auditors article.

Recommendation: If the EVault is deployed on L2, it is recommended to use CREATE2 instead of CREATE to deploy the proxies to deterministic addresses.

Euler: Acknowledged, no fix. Users concerned with reorgs should wait for sufficient blocks to be mined.

Spearbit: Acknowledged.

5.5.4 Naming improvement suggestions

Severity: Informational

Context: EulerSavingsRate.sol#L14-15, ESynth.sol#L63

Description:

- REENTRANCYLOCK__UNLOCKED and REENTRANCYLOCK__LOCKED do not follow the common Open Zeppelin naming case (NOT_ENTERED/ENTERED), additionally it is a long and redundant naming. Consider the following alternative namigs: UNLOCKED/ LOCKED, REENTRANCY_UNLOCKED/REENTRANCY_LOCKED instead.
- burn function uses an input address, not msg.sender. Therefore, a more descriptive naming could be burnFrom.

Recommendation: Consider applying the aforementioned suggestions.

Euler: Fixed in commit 83105b18.

Spearbit: The commit partially implements the recommendations. The burn function has not been renamed.

Euler: Will keep as is to retain compatibility with Chainlink CCIP.

5.5.5 The flashLoan function doesn't emit a dedicated FlashLoan event

Severity: Informational

Context: Borrowing.sol#L164-L177

Description: The flashLoan function currently only emits the ERC20:Transfer events, which are triggered in the ERC20 transfers.

Recommendation: To improve monitoring of the flashLoan related functionality, you consider emitting a dedicated FlashLoan event which will track the amount in asset units and the amount repaid.

Euler: Acknowledged, no fix. The implementation is designed to be as gas efficient as possible to compete with other providers.

Spearbit: Acknowledged.

5.5.6 Missing EVC() getter on ERC20Collateral and EulerSavingsRate

Severity: Informational

Context: ERC20Collateral.sol#L15, EulerSavingsRate.sol#L49

Description: The ERC20Collateral and EulerSavingsRate are EVC-compatible but, on-chain, one cannot see what EVC these contracts are using as they are missing an EVC() getter to return the internal evc address. It's currently unclear what EVC deployment these contracts that can be used as collateral are compatible with which could lead to misconfigurations.

Recommendation: Consider exposing the internal evc address through a getter.

Euler: Fixed in PR 156 of the EVC repo.

Spearbit: The issue has been mitigated by the EVC PR 156.

5.5.7 Consider not emitting the Approval event in BalanceUtils.decreaseAllowance following the same behavior of OZ ERC20

Severity: Informational

Context: BalanceUtils.sol#L117

Description: The OpenZeppelin implementation of the ERC20 standard emits the Approve event only when the allowance is directly modified via approve.

Recommendation: Euler should consider following the same behavior of the OZ ERC20 implementation, avoiding emitting the Approve event in the BalanceUtils.decreaseAllowance function

Euler: Fixed in PR 212.

Spearbit: The issue has been mitigated by PR 212.

5.5.8 Vault.skim should follow the same operation order of Vault.deposit

Severity: Informational

Context: Vault.sol#L194-L195

Description: The skim operation is equivalent to a deposit operation without the need to "pull" the assets to be deposited from the sender given that those funds have been already deposited into the vault.

With such a premise, the skim function should follow the same order of operations that the deposit function is performing when it executes finalizeDeposit(...).

Recommendation: Euler should:

1) Update the vaultStorage.cash **before** the increaseBalance function like finalizeDeposit is doing.
2) Add an inline comment that explains that this is performing finalizeDeposit without the pullAsset logic that pulls assets from the sender.

Euler: Fixed in PR 166.

Spearbit: The issue has been mitigated by PR 166.

5.5.9 liabilityValue does not need to be re-calculated Liquidation.calculateMaxLiquidation

Severity: Informational

Context: Liquidation.sol#L145-L149

Description: The calculateMaxLiquidation function in the Liquidation modules is recalculating the liabilityValue like this

```
uint256 liabilityValue = liqCache.liability.toUint();
if (address(vaultCache.asset) != vaultCache.unitOfAccount) {
    liabilityValue =
        vaultCache.oracle.getQuote(liabilityValue, address(vaultCache.asset), vaultCache.unitOfAccount);
}
```

But such value has been already calculated at the very beginning of the function and has been stored in liquidityLiabilityValue returned by the calculateLiquidity(...) execution.

Recommendation: Euler should remove the redundant code and initialize liabilityValue with liquidityLiabilityValue if needed or use directly liquidityLiabilityValue

Euler: Fixed in PR 167.

Spearbit: The issue has been mitigated by PR 167.

5.5.10 Consider improving `clearLTV` and `setLTV(..., ltv=0, ...)` documentation

Severity: Informational

Context: Governance.sol#L231-L240, Governance.sol#L205-L229

Description: While it's clear when the `clearLTV` should be called (given the Natspec documentation), it's particularly clear the scenarios for which the governance should call `setLTV(collateral, 0, rampDuration > 0)` or `setLTV(collateral, 0, rampDuration = 0)` and not `clearLTV` or vice versa, and which are the specific consequences of those three scenarios and what happens when the LTV will reach `targetValue = 0`.

Recommendation: Euler should consider providing clear guidelines on when each function should be used and which are the consequences of setting `LTV=0` with or without a ramp duration.

Euler: Fixed in PR 214.

Spearbit: The issue has been mitigated by PR 214.

5.5.11 `Governance.clearLTV` should revert if the `LTV` has never been configured

Severity: Informational

Context: Governance.sol#L231-L240

Description: The `clearLTV` function should be callable only if an `LTV` has been configured and initialized.

Recommendation: The transaction should revert if `vaultStorage.ltvLookup[collateral].initialized == false`

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5.12 Enforce `EVC` compatibility on new `collateral` added to `EVK` via `setLTV`

Severity: Informational

Context: Governance.sol#L208

Description: The current documentation of `EVC` and `EVK` white paper describe the collateral as "the address of another vault" but the `Governance.setLTV` function does not perform any sanity check to enforce such requirements.

Recommendation: Euler should:

- Require that the configured `collateral` does indeed support the EVC platform and that the `evc` address returned by the collateral is equal to the one configured inside the vault.
- Update the `EVC`, `EVK` and all the white papers to clarify what the collateral should be. Euler has clarified that the collateral does not need to be an `EVK` vault specifically, but it's required to be EVC-compatible. This fact is not reflected in the current documents.

Euler: The recommended sanity check is insufficient to determine whether a given vault is safe to use as collateral for another vault. Consequently, the protection against setting the LTV for a contract, which returns a different address via the `EVC` function, offers only a false sense of security and does not enhance the overall safety of the system.

Things to note:

1) There is no requirement for any EVC-compatible contract to have a function that returns the address of the EVC it internally uses. Moreover, the implementation of such a function does not constitute EVC compatibility. From a security perspective, it is merely a nice-to-have function.
2) The EVC documentation website adequately documents all the requirements that a vault must meet to be considered EVC-compatible. The Vault Implementation Considerations and the Vault Implementation Guide sections provide extensive considerations and an explicit list of steps necessary for a vault to achieve EVC compatibility, function by function.

3) For a vault to be valid collateral, apart from the risk management perspective, it is technically sufficient for it to be EVC-compatible, which we consider self-explanatory given that the EVK is built on top of the EVC. This includes using the EVC execution context's stored in `onBehalfOfAccount` (supporting sub-accounts and `controlCollateral`) and scheduling account status checks for addresses whose health may be negatively affected by an operation. Both requirements are covered in the Vault Implementation Considerations and Vault Implementation Guide.

4) Finally, the Untrusted Collaterals section of the EVK white paper thoroughly describes the trust assumptions for collateral vaults, although it could be refined.

All things considered, there is no need to perform the sanity check in the `Governance.setLTV` as it only provides a false sense of security. However, the Untrusted Collaterals section of the white paper could be refined to explicitly mention that:

1) A collateral vault must use the same EVC instance as the controller.

2) A collateral vault must be EVC-compatible as per Vault Implementation Considerations of the evc.wtf.

Spearbit: Acknowledged.

5.5.13 Consider including the `initialized` value in the `GovSetLTV` to track if the configured `LTV` is new or not

Severity: Informational

Context: Governance.sol#L37-L39

Description: The `originalLTV == 0` information is not enough to know whether an LTV is new or not. Euler should consider including the `initialized` value in the `GovSetLTV` event to understand if the event has been emitted for a new or already configured `LTV`

Recommendation: Consider including the `initialized` inside the `GovSetLTV` event.

Euler: Fixed as recommended in PR 168.

Spearbit: Verified.

5.5.14 Consider allowing the user to disable the balance forwarder flag even when `balanceTracker` is not configured

Severity: Informational

Context: BalanceForwarder.sol#L39-L52

Description: Considering the `EVault` could break if users have enabled balance forwarding and the `balanceTracker` has been upgraded to `address(0)` scenario, a user could personally fix the operation by disabling the Balance Forwarder flag even if `balanceTracker` is equal to `address(0)` and the flag was previously enabled.

Recommendation: Euler should consider allowing the user to disable the Balance Forwarder flag even when `address(balanceTracker) == address(0)`. Such feature needs a refactor of `disableBalanceForwarder` that must not call `balanceTracker.balanceTrackerHook` if the tracker is not configured.

Euler: The `balanceTracker` address is an immutable vault parameter that gets specified on the vault implementation contract deployment. This address can only change if the EVK vault factory governor deploys a new implementation contract which will affect the beacon proxies pointing to it.

Indeed, in such circumstances, the address can change from non-zero to zero which may prevent some users, had they enabled balance forwarding, from interacting with the vault. However, it must be noted that upgradable smart contracts introduce such risk by definition and it is not unique to the EVK. An irresponsible upgrade admin can upgrade a contract to any implementation and cause a denial of service for their users. A defensive coding that prevents against upgrade admin irresponsibility is not an industry standard approach therefore no changes are required.

The balanceTracker address is an immutable vault parameter that gets specified on the vault implementation contract deployment. This address can only change if the EVK vault factory governor deploys a new implementation contract which will affect the beacon proxies pointing to it.

Euler: Indeed, in such circumstances, the address can change from non-zero to zero which may prevent some users, had they enabled balance forwarding, from interacting with the vault. However, it must be noted that upgradable smart contracts introduce such risk by definition and it is not unique to the EVK. An irresponsible upgrade admin can upgrade a contract to any implementation and cause a denial of service for their users. A defensive coding that prevents against upgrade admin irresponsibility is not an industry standard approach therefore no changes are required.

Spearbit: Acknowledged.

5.5.15 Improve the documentation about the `Oracle` in the `EVK` white paper

Severity: Informational

Context: Euler Vault Kit white paper

Description: The `Oracle` component is a crucial part of the `EVK` protocol and should be properly documented in both the codebase and `EVK` whitepaper. Some of the questions that a user, integrator or deployer could have (but not limited to) are:

- Can we assume that the `EulerRouter` (the oracle returned by `metadata()`) has been correctly configured?
- Assuming that the `EulerRouter` has been correctly configured, can the `oracle.getQuote` revert?
- Assuming that the `EulerRouter` has been correctly configured, can the `oracle.getQuote` return `0`?
- Other assumptions or non-assumptions that can be made/not made?

Recommendation: Euler should improve the documentation about the `Oracle` component inside the `EVK` codebase and `EVK` whitepaper.

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5.16 Improve the white paper Interest Overflow section, including the side effects of `RPow.rpow` overflow scenario

Severity: Informational

Context: Cache.sol#L79-L89, Cache.sol#L109, Interest Overflows

Description: When the `RPow.rpow` overflows, the `newInterestAccumulator` is not updated and will use the old cached value. This means that interest is not accrued and `newTotalBorrows` will remain equal to `vaultCache.totalBorrows`.

If `newTotalBorrows` is `<= MAX_SANE_DEBT_AMOUNT` vault will update `lastInterestAccumulatorUpdate` to `uint48(block.timestamp)` anyway (unlike the case when `newTotalBorrows` overflows the sane amount). As a consequence, the whole interest accrued during `deltaT` time will be lost and "reset".

Recommendation: Euler should Improve the white paper Interest Overflow section, including the side effects of `RPow.rpow` overflow scenario

Euler: Acknowledged, no fix. The white paper was updated to better cover this and similar effects.

Spearbit: Acknowledged.

5.5.17 Consider enhancing the Base.isOperationDisabled and all the max* vault functions documentation

Severity: Informational

Context: Base.sol#L92-L97

Description: An EVault operation can be considered disabled in two cases:

1) The hookedOps flag of the operation is set and the hookTarget == address(0).
2) The hookedOps flag of the operation is set, the hookTarget != address(0) and the hookTarget reverts internally in a "sane" way to disclose the disablement of the operation.

The second case can't be evaluated directly by the protocol without executing directly the operation, and it's out of scope of current usage of isOperationDisabled inside EVault that optimistically evaluates the maximum number of tokens that can be deposited/minted/withdrawn/redeemed.

Given that the second scenario could happen, such an eventuality should be documented in both the isOperationDisabled and all the max* vault functions to warn the user that the returned value is an optimistic evaluation.

Recommendation: Euler should enhance the documentation of those functions or evaluate the refactoring of the logic to simulate the response of the hookTarget to provide a more reliable answer in the max* vault functions.

Euler: We acknowledge the issue. We consider the issue sufficiently documented in the white paper.

Spearbit: Acknowledged.

5.5.18 BorrowingUtils.transferBorrow and BalanceUtils.transferBalance do not handle correctly the self-transfer case

Severity: Informational

Context: BalanceUtils.sol#L70-L96, BorrowUtils.sol#L72-L95

Description: In both functions, when from == to the account balance will be wrongly updated. With the current codebase, this is not a security issue because the callers of this function will revert when the user tries to self-transfer to itself, but it's a good practice to avoid these kinds of mistakes directly to ensure that future changes to the protocol won't fall into this problem.

Recommendation: Euler should refactor both BorrowingUtils.transferBorrow and BalanceUtils.transferBalance to avoid miss accounting the user balance when from == to. A possible fix could be to update the from storage and load the to storage after.

Euler: Fixed in PR 181.

Spearbit: The issue has been mitigated by PR 181.

5.5.19 EVault could break if users have enabled balance forwarding and the balanceTracker has been upgraded to address(0)

Severity: Informational

Context: BalanceUtils.sol

Description: Every time the user's balance changes or the balance tracking flag is changed to true or false the balanceTracker.balanceTrackerHook hook will be triggered.

While in the BalanceForwarderModule module the flag can be changed only if the balanceTracker has been configured, and for such reason, the hook can be triggered only in a non-reverting environment, in the BalanceUtils the value of balanceTracker is never sanity checked, and the hook is always triggered if the user has enabled the flag.

If the user has enabled the flag and the EVault is upgraded to a vault that has the balanceTracker set to address(0) any interaction with the vault that updates the user's balance will revert, resulting in a broken vault.

Recommendation: Euler should ensure that the `balanceTracker.balanceTrackerHook` hook can be called only if the `balanceTracker` is not `address(0)`.

Euler: The `balanceTracker` address is an immutable vault parameter that gets specified on the vault implementation contract deployment. This address can only change if the EVK vault factory governor deploys a new implementation contract which will affect the beacon proxies pointing to it.

Indeed, in such circumstances, the address can change from non-zero to zero which may prevent some users, had they enabled balance forwarding, from interacting with the vault. However, it must be noted that upgradable smart contracts introduce such risk by definition and it is not unique to the EVK. An irresponsible upgrade admin can upgrade a contract to any implementation and cause a denial of service for their users. A defensive coding that prevents against upgrade admin irresponsibility is not an industry standard approach therefore no changes are required.

Spearbit: Acknowledged.

5.5.20 Zero address returned from `MetaProxyDeployer` is not explicitly handled in `GenericFactory`

Severity: Informational

Context: GenericFactory.sol#L86, MetaProxyDeployer.sol#L10

Description: The `MetaProxyDeployer` uses the `CREATE` opcodes to create a new instance, which returns a zero address and does not revert if the deployment fails.

If a zero address is returned, execution will revert during `IComponent(proxy).initialize(msg.sender)`.

Recommendation: It is recommended to explicitly handle the zero-address scenario with a custom error to improve error handling on the frontend.

Euler: Fixed in commit c3726682.

Spearbit: The issue has been mitigated by commit c3726682.

5.5.21 `IBalanceTracker` natspec documentation should be improved

Severity: Informational

Context: BalanceForwarder.sol#L34, BalanceForwarder.sol#L49, IBalanceTracker.sol#L13

Description: Unlike all the "normal" operations inside the various modules where the balance is always increased or decreased compared to the value before the operation, the `BalanceForwarderModule.enableBalanceForwarder` and `BalanceForwarderModule.disableBalanceForwarder` will execute `balanceTracker.balanceTrackerHook` with "special" values.

- `BalanceForwarderModule.enableBalanceForwarder` will always report the current user's balance, unchanged.
- `BalanceForwarderModule.disableBalanceForwarder` will always report `0` even if the balance of the user is greater than zero.

On top of these "special" values, the contract that implements the `IBalanceTracker` interface should be aware that `balanceTrackerHook` could be called multiple times, even inside the same block, given that `enableBalanceForwarder` and `disableBalanceForwarder` have no restrictions.

These custom behaviors should be documented by the `IBalanceTracker` interface for the `newAccountBalance` parameter.

Recommendation: Euler should consider documenting these custom behaviors and values inside the `IBalanceTracker` interface.

Euler: Fixed in PR 169 as recommended.

Spearbit: Verified.

5.5.22 Inaccurate `Deposit` event can be emitted during the `skim` function

Severity: Informational

Context: Vault.sol#L176-L198

Description: When the `skim` function is called, the `Deposit` event is emitted with the `onBehalfOfAccount` as the sender. However, the `onBehalfOfAccount` might not be responsible for the excess asset balance.

Recommendation: It is recommended to document that during the `skim` function, the `sender` field may be inaccurate.

Euler: Acknowledged, no fix. We consider the account claiming the excess tokens as the actual sender of the deposit. It is assumed that if they were able to orchestrate the tokens arrive in the vault, it's equivalent to holding them directly, or sufficient for the purpose of the event.

Spearbit: Acknowledged.

5.5.23 Consider adding to the `PegStabilityModule` utility functions that allow to preview the amount of asset received with a swap

Severity: Informational

Context: PegStabilityModule.sol

Description: `PegStabilityModule` allows users to exchange `ESynth` asset for an `underlying` asset (less a fee) and vice versa. It would be helpful for the users, before executing the real exchange, to preview the returning amount given:

- the fee to be applied.
- The available liquidity of the assets:
  - Available `underlying` balance if the users want to swap `ESynth` for `underlying`.
  - Available `PegStabilityModule` minting capacity if the users want to swap `underlying` for `ESynth` (that must be minted new).
- Possible user restriction when the user's `ESynth` asset are exchanged (burned) for `underlying`. The `ESynth.burn` function calls `ERC20Collateral._update` which will require an account status check for the user who has called the `PegStabilityModule`. If the user has enabled a controller and is unhealthy, the transaction will revert.

Recommendation: Euler should consider implementing utility functions that allow the user to preview the maximum amount that the user can exchange in an operation given fees, available liquidity of the assets and the user's restriction on EVC.

Euler: Acknowledged. Although possibly convenient for integrators these things can either be calculated by integrators themselves or be later calculated in a helper contract if there's a demand to do so.

For simplicity's sake we'll keep it as is.

Spearbit: Acknowledged.

5.5.24 If `gulp` is never called, available interest is not accounted and accrued and withdrawing users won't receive deserved interest

Severity: Informational

Context: EulerSavingsRate.sol#L134-L148

Description: `gulp` is the ESR mechanism that starts the accrual of the amount of asset that has been sent to the ESR by an external entity. Once gulp is called, such amount is added to `esrSlot.interestLeft` and `esrSlot.interestSmearEnd` is resetted to `block.timestamp + INTEREST_SMEAR`.

If no one calls `gulp`, the interest won't start accruing even if it has been already deposited in the ESR module, and users who withdraw from the ESR module won't receive the deserved interest that they should receive:

1) Alice deposits `10e18`.
2) `10e18` interests are sent to the ESR.
3) Alice waits 2 weeks.
4) Alice withdraws, thinking that she will get `20e18`, but she will only get back her `10e18` initially deposited.

Recommendation: Unfortunately, as already mentioned in the issue "`EulerSavingRate` `gulp` can delay the full accrual of the user's interest", calling `gulp` automatically when a user's operation happens is not a viable option because it could risk doing more harm than good. Euler should consider refactoring and reimplementing the interest accrual of the ESR module to avoid such issue.

Euler: Large depositors are incentivized to periodically call gulp if that would mean their effective APR goes up we expect this to happen enough.

If this assumption proves to not hold a remediation would be to attach a keeper to periodically call gulp.

Again I disagree with severity, its "informational" at best, no funds at risk and in practice the issue will most likely be non existent due to the economic incentives at play.

The current system is low complexity code wise and feature complete. Introducing more complexity to fix an issue which is not really an issue would be unwise from a security perspective (from my point of view)

Will keep as is.

Spearbit: Acknowledged.

5.5.25 `EulerSavingsRate` uses default virtual shares

Severity: Informational

Context: EulerSavingsRate.sol#L13, ERC4626.sol#L225-L234

Description: The `EulerSavingsRate` is itself an ERC4626 vault that can directly be used as collateral in the EVC, without first wrapping the shares in an escrow vault. Therefore, it should implement the same parameters as the eVault to get the same level of price share manipulation resistance.

Recommendation: Consider adjusting the virtual shares in the ERC4626 OZ conversion.

Euler: Fixed in commit 51c2133d.

Spearbit: The provided commit mitigates the issue.

5.5.26 `maxRedeemInternal` could be private

Severity: Informational

Context: Vault.sol#L228-L248

Description: The `maxRedeemInternal` function could be `private` instead of `internal`, similar to `maxDepositInternal`.

Recommendation: Consider changing its visibility to match `maxDepositInternal`.

Euler: Fixed in PR 155.

Spearbit: The provided PR mitigates the issue.

5.5.27 `RiskManager.checkAccountStatus` will be executed even if the user has interacted with a non-collateral asset

Severity: Informational

Context: Multiple instances across `EVK`, `ESynth`, `PegStabilityModule` and `EulerSavingRate`

Description: When an `account` (directly or "indirectly" on behalf of him/her) performs an operation (`transfer`, `redeem`, `withdraw`) that could decrease his/her health factor, the `EVK` ecosystem (`EVault`, `ESynth`, `PegStabilityModule`, `EulerSavingRate`) will require the `EVC` to perform a check at the end of the `EVC` `call` or `batch` flow if the user has enabled a `controller`.

At the end of the flow, `EVC` will call `RiskManager`.checkAccountStatus‘ which will always revert if the user is unhealthy. If the user has a controller enabled, this logic will always be executed, without considering which was the asset that was interacted with.

This means that `checkAccountStatus` will be invoked even if the user had not enabled the asset as collateral. When an asset is not enabled as collateral, it means that it cannot influence the user's health factor and should be allowed to be transferred or withdrawn freely without triggering a health check status.

Because of the EVK/EVC logic and the current behavior, we have the following negative side effects:

- If the user is unhealthy, the transaction will revert even if the user tries to `transfer`, `redeem` or `withdraw` a non-collateral asset (that cannot decrease the HF furthermore).
- `transfer`, `redeem` or `withdraw` of a non-collateral asset will consume more gas than it should because of the check-account-status additional logic.

The above negative side effects will be applied in all these cases:

- EVK `transfer`, `transferFrom`, `withdraw` and `redeem`.
- `ESynth` `transfer` and `transferFrom`.
- Swapping `ESynth` for underlying on the `PegStabilityModule` via the `swapToUnderlyingGivenIn` and `swapToUnderlyingGivenOut`.
- `EulerSavingRate` `transfer`, `transferFrom`, `withdraw` and `redeem`.

Note that this behavior differs from what a normal user is used to with other lending protocols. Usually, an operation that involves a non-collateral asset can be performed freely without any restriction related to the user's health factor and will consume less compared to an operation on a collateral asset.

Recommendation: Euler should consider the viability of a refactor of this logic to allow the user to perform those operations without any restriction and with a lower gas cost if the involved asset is not a collateral asset.

If this is not an option, Euler should document and warn the user about this behavior, given that the experience with Euler will be different compared to how other lending protocol works.

Euler: We acknowledge the issue.

The behavior is intended. From the technical perspective, changing it would require changing some of the security-critical code, which would increase complexity and attack surface. On the other hand, we consider the vault to be in its rights to motivate accounts with unhealthy borrows to mitigate the situation. Blocking withdrawals of non-collateral assets is one such motivation.

A section of white paper was added to inform users about the behavior.

Spearbit: Acknowledged.

5.5.28 Observed values related to `interestAccumulator` can drop once `loadVault()` handles overflows

Severity: Informational

Context: Cache.sol#L82, RiskManager.sol#L76

Description: Some functions only load the vault and update it in memory, without writing the updated data back to storage. Combined with the fact that old interest accumulators (and by extension old `totalBorrows` and `totalShares`) are used in case the new interest accumulator would overflow, it can lead to the situation that view functions or `checkAccountStatus` use values that first rise, but then suddenly drop once the accumulator overflows.

Recommendation: Consider documenting this behavior in `loadVault()` and using `updateVault()` in non-view functions like `checkAccountStatus`. Alternatively, instead of using the last accumulator value, consider using a *max* accumulator value (that does not overflow when used in other computations). This would prevent user debt from suddenly dropping.

Euler: Acknowledged, no fix. The white paper was updated to better cover this and similar effects.

Spearbit: Acknowledged.

5.5.29 Liquidations that don't repay debt still emit borrow events

Severity: Informational

Context: Liquidation.sol#L177

Description: If the `violator` is healthy, liquidation continues with a no-op. The `transferBorrow(vaultCache, liqCache.violator, liqCache.liquidator, liqCache.repay);` code is always executed and would emit events transferring 0 assets.

Recommendation: Consider guarding the borrow transfer by checking `if (liqcCache.repay > 0)`.

Euler: Fixed as recommended in PR 172. Pulling dust is a valid use case which, allows debt socialization. But also, there should never be just debt dust on the account, repay and debt transfer logic should prevent it.

Spearbit: Verified.

5.5.30 Ambiguous return parameters for `loop` / `deloop`

Severity: Informational

Context: Borrowing.sol#L133-L136, Borrowing.sol#L102

Description: The `deloop` function can readjust the `assets` parameter in case `assets > owed`. A caller must not believe that the returned `shares` are equivalent to the `amount` parameter they used (even if `amount != uint256.max`).

Recommendation: Consider returning both `(uint256 assets, uint256 shares)` for `loop` and `deloop`.

Euler: Fixed as recommended in PR 173.

Spearbit: Verified.

5.5.31 Unclear usecase for `loop`

Severity: Informational

Context: Borrowing.sol#L96, Governance.sol#L207-L208

Description: The Euler protocol does not allow self-collateralization (using the same asset as collateral that is borrowed). Therefore, any `loop` call can be implemented with a single round of a `borrow(account); deposit(sharesReceiver)` sequence in a batch.

Recommendation: Consider removing functions that don't have a clear use case and can be represented by simple batch actions.

Euler: Removed in PR 200.

Spearbit: Euler has removed the `loop` functionality and has renamed the `deloop` one to `repayWithShares`, updating all the relevant parts of the codebase.

5.5.32 Caching of interest rate could lead to issues for non-pure IRMs

Severity: Informational

Context: RiskManager.sol#L86

Description: The interest rate is retrieved from the IRM once in the `checkVaultStatus` function. The result is then cached to storage. This cached rate will be used in the subsequent vault interactions (that can happen at different blocks). If an advanced IRM depends on `block.timestamp` or other derived on-chain state, the interest rate can change but it will not be used for the active vault interactions.

Recommendation: IRMs should be pure functions that depend only on the input to its `computeInterestRate(address vault, uint256 cash, uint256 borrows)` interface.

Euler: Acknowledged. I expect IRMs will usually be pure functions, but the system should work even if not. Yes, the rates could be stale, but users can re-target them as frequently as they want using the `touch()` function (if rates are too low, I'd expect depositors to re-target, otherwise borrowers).

Spearbit: Acknowledged.

5.5.33 Interest rate will be underestimated due to keeping utilisation constant

Severity: Informational

Context: Cache.sol#L80

Description: The interest rate compounds every second with a cached interest rate (per second). However, the IRM would quote a higher interest rate if the compounding happened every second (or block) as the `utilisation = totalBorrows / (totalBorrows + cash)` increases.

Recommendation: In active vaults, the difference should be negligible.

Euler: We acknowledge the issue and agree with the recommendation: `In active vaults, the difference should be negligible.`

Spearbit: Acknowledged.

5.5.34 Forgiving vault checks would end up with lingering snapshot

Severity: Informational

Context: RiskManager.sol#L109

Description: In case the EVK implements an `EVC.forgiveVaultStatusCheck` call, the vault snapshot would not be cleared as it is only cleared in the `checkVaultStatus` callback. The next vault interaction will *not* overwrite the snapshot, instead, it will perform the check on an outdated, lingering snapshot.

Recommendation: In case the EVK implements an `EVC.forgiveVaultStatusCheck` call, a new vault snapshot clearing behavior also needs to be implemented.

Euler: Comments improved in PR 174.

Spearbit: Verified.

5.5.35 Inconsistent rounding for `yield = repay / discount` liquidation computation

Severity: Informational

Context: Liquidation.sol#L151-L160, Liquidation.sol#L162-L163

Description: For liquidations, the repaid amount relates to the seized collateral (yield) by `yield = repay / discount`, rounding down the yield. However, if the violator's collateral balance is less than the max yield, the entire collateral balance is seized and the repaid amount is readjusted, rounding down the repaid amount this time.

Recommendation: For consistency, consider always rounding up the repaid values if computed from yield (and because eliminating more debt is generally better for the protocol).

Note that additional arguments are needed to ensure that the final `liqCache.repay` amount is always less than the user's total debt assets `liqCache.liability`.

```
/* Note: shows the current code */
// rounds down yield here
uint256 maxRepayValue = liabilityValue;
uint256 maxYieldValue = maxRepayValue * 1e18 / discountFactor;

if (collateralValue < maxYieldValue) {
    /* currently: maxRepayValue' = floor(collateralValue * discountFactor / 1e18)
                <= floor(maxYieldValue * discountFactor / 1e18)
                = floor(floor(maxRepayValue * 1e18 / discountFactor) * discountFactor / 1e18)
                <= maxRepayValue = liabilityValue
    */
    /* rounding up still keeps: maxRepayValue' <= liabilityValue
        maxRepayValue' = ceil(collateralValue * discountFactor / 1e18)
        <= ceil(maxYieldValue * discountFactor / 1e18)
        = ceil(floor(maxRepayValue * 1e18 / discountFactor) * discountFactor / 1e18)
        <= maxRepayValue = liabilityValue
    */
    maxRepayValue = collateralValue * discountFactor / 1e18;
    maxYieldValue = collateralValue;
}

// could round up here too: maxRepayValue <= liabilityValue, therefore liqCache.repay <=
↪  liqCache.liability
liqCache.repay = (maxRepayValue * liqCache.liability.toUint() / liabilityValue).toAssets();
liqCache.yieldBalance = maxYieldValue * collateralBalance / collateralValue;
```

Euler: Acknowledged, no fix. The algorithm to calculate the yield and repay is entirely arbitrary, as is the derivation of the discount for the liquidator. The actual values depend much more on the market prices of collateral vs liability, than on rounding directions. For simplicity, we prefer to keep the code as is.

Spearbit: Acknowledged.

5.5.36 IPriceOracle is out of sync with euler-price-oracle repo

Severity: Informational

Context: IPriceOracle.sol#L5

Description: The interface appears to be out of sync with the euler-price-oracle repository and is inconsistent. It includes features like the name getter, which isn't present in the original. Moreover, defined errors are only used in MockPriceOracle.sol during testing.

Recommendation: Consider removing this interface and using the official one from the euler-price-oracle project to maintain consistency and prevent redundancy. Alternatively, keep the interfaces aligned between both repositories if dependency concerns prevent the direct use of the original.

Euler: Fixed in commit a2349358.

Spearbit: Euler has updated the IPriceOracle interface, syncing it with the one from the euler-price-oracle project but has not added the direct integration. The recommendations have been implemented in commit a2349358.

5.5.37 Unused reentrancy lock in BaseProductLine

Severity: Informational

Context: BaseProductLine.sol#L23

Description: reentrancyLock is not used actively at BaseProductLine nor on any contract that inherits from it, therefore it should be removed

Recommendation: Remove unnecessary elements from the code in order to decrease complexity and improve readability.

Euler: Acknowledged. We decided to remove this contract entirely.

Spearbit: Acknowledged.

5.5.38 Unused logic and confusing event in setVaultInterestFeeRange and setVaultFeeConfig

Severity: Informational

Context: ProtocolConfig.sol#L180, ProtocolConfig.sol#L201

Description: When exists_ is false, updating _interestFeeRanges[vault] with a non-default value and emitting an event can be confusing. It forces the caller to provide valid minInterestFee_ and maxInterestFee_ values, which won't be used when _interestFeeRanges is retrieved because it would be necessary to re-execute setVaultInterestFeeRange(vault, true, ...) to enable it with proper interest fees.

Recommendation: Instead of updating _interestFeeRanges[vault] with a non-default value, call delete _interestFeeRanges[vault] and emit a separate event. This will simplify the logic and prevent the need to pass unused values.

Euler: Acknowledged, no fix. Since it's a privileged function, we expect the inputs to be properly constructed.

Spearbit: Acknowledged.

5.5.39 `uint` caps version is not consistently used

Severity: Informational

Context: Base.sol#L85

Description: `vaultCache` uses the `uint256` version of caps rather than the `uint16` version from `vaultStorage` (`AmountCap`):

- If `vaultCache.supplyCap != type(uint256).max`, it should be `<= 2 * MAX_SANE_AMOUNT` (see `Governance.setCaps` checks).
- If `vaultCache.borrowCap != type(uint256).max`, it should be `<= MAX_SANE_AMOUNT` (see `Governance.setCaps` checks).

Using the `type(uint256).max` value directly for the "no cap" scenario would be cleaner

Recommendation: Update the logic to consistently use `type(uint256).max` for both `supplyCap` and `borrowCap` in the "no cap" scenario.

```diff
- if ( !vaultCache.snapshotInitialized
-    && (vaultCache.supplyCap < type(uint256).max || vaultCache.borrowCap < type(uint256).max)
+ if ( !vaultCache.snapshotInitialized
+    && (vaultCache.supplyCap != type(uint256).max || vaultCache.borrowCap != type(uint256).max)
  ) {
  // code
  }
```

Euler: Fixed in PR 175 as recommended.

Spearbit: Verified.

5.5.40 `pushAssets` would benefit from extra documentation

Severity: Informational

Context: AssetTransfers.sol#L28

Description: The handling of sub-accounts through the `EVC` flag would benefit from extra clarifications.

1. `CFG_EVC_COMPATIBLE_ASSET` **flag:** This flag should be true if the vault's underlying asset is another vault or an EVC-compatible ERC20Collateral.
2. `pushAssets` **function checks:** The whole idea of the checks is to protect users from mistakenly setting a sub-account (non-zero one) as `receiver` in functions that send tokens out (`withdraw`, `redeem`, `borrow`). If a regular asset is sent to a sub-account it would effectively be lost, since the private keys are not known. It's only EVC that understands that sub-accounts have owners and only assets that authenticate through EVC can safely accept `receiver` that is a sub-account.
3. **Example USDC Vault transfer cases:**
   - **Case 1:** transfer to non-registered EVC account → `success`. It is allowed just because it is not known if it's a sub-account or owner.
   - **Case 2:** transfer to a registered EVC account that is equal to the owner of the account → `success`. You know that `account` can indeed interact with EVC so it means that it's an EOA/contract that will be able to interact eventually with the ERC20. This is not entirely true, you know that `account` can interact with EVC, but you don't know if it can interact with the `asset`. `account` in this case could be `!= originalCaller`.
   - **Case 3:** transfer to a registered EVC account that's not owned by the caller, with a high probability that the asset is unrecoverable as that `owner` probably does not own the private key of the `account` to later on interact with `asset`.

Recommendation: Consider adding more explicit documentation regarding, `CFG_EVC_COMPATIBLE_ASSET`, the checks in the `pushAssets` and a practical example.

Euler: Fixed in PR 196.

Spearbit: Verified.

5.5.41 `calculateDTokenAddress` may fail if anything changes in the future code

Severity: Informational

Context: BorrowUtils.sol#L150

Description: On `calculateDTokenAddress`, `mstore8(0x34, 0x01)` is true because `InitializeModule.initialize` creates a new `DToken` as the first contract deployed by the vault.

If anything changes in the future code of `InitializeModule.initialize` or during the initialization flow (contract deployed before `DToken`) the `0x01` value should be adjusted accordingly.

Recommendation: Improve the documentation and keep an eye on this value in future updates.

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5.42 `checkLiquidation` doesn't revert if `violator` has no debt or has more collateral than `debt`

Severity: Informational

Context: IEVault.sol#L244

Description: `checkLiquidation` function does not revert if the `violator` has no debt or has more collateral than debt (in value). In that case, both `maxRepay` and `maxYield` will be equal to zero.

Recommendation: Consider being explicit about this behavior. Some integrators could expend a revert when the liquidation can't be performed. Other protocols behave differently, reverting when the liquidation can't be performed.

Euler: Fixed in commit 197c670e.

Spearbit: Euler has documented the behavior in commit 197c670e.

5.5.43 Inconsistency in `CONTROLLER_NEUTRAL_OPS`

Severity: Informational

Context: Constants.sol#L55-L56

Description: `CONTROLLER_NEUTRAL_OPS` is a constant that incorporates different OPs for later use:

```
uint32 constant CONTROLLER_NEUTRAL_OPS = OP_DEPOSIT | OP_MINT | OP_WITHDRAW | OP_REDEEM | OP_TRANSFER |
↪  OP_SKIM | OP_REPAY | OP_DELOOP | OP_CONVERT_FEES | OP_FLASHLOAN | OP_TOUCH;
```

However, it is noticed that `OP_VAULT_STATUS_CHECK` should be one of the controller-neutral operations because it's unrelated to a specific account's borrowing state.

It doesn't matter in the actual state of the codebase as it's not used via `initOperation` (the only place `CONTROLLER_NEUTRAL_OPS` is used) but so does `OP_FLASHLOAN` and it is in this list.

Recommendation: Incorporate `OP_VAULT_STATUS_CHECK` to the `CONTROLLER_NEUTRAL_OPS` list.

Euler: Fixed in PR 176.

Spearbit: Verified.

5.5.44 Casting to the same type is redundant and adds verbosity

Severity: Informational

Context: EVCClient.sol#L113

Description: Casting a type to the same type is redundant and adds verbosity. In `EVCClient`, `evc` (of `IEVC` type) is cast to `IEVC` again.

Recommendation: Remove redundant casts:

```
- address[] memory controllers = IEVC(evc).getControllers(account);
+ address[] memory controllers = evc.getControllers(account);
```

Euler: Fixed in commit c96862ca.

Spearbit: Verified.

5.5.45 `validate` should be moved to `Types.sol`

Severity: Informational

Context: Types.sol#L80

Description: `ConfigAmountLib.validate` is only used in `Types.sol` and therefore, the logic can be moved to `Types.sol` in order to enhance simplicity and coherence.

Recommendation: Remove `validate` from `ConfigAmountLib` and add it to `Types`

Euler: Fixed in PR 186.

Spearbit: Verified.

5.5.46 Consistently use `toUint` rather than `unwrap`

Severity: Informational

Context: ConfigAmount.sol#L14

Description: `Assets`, `Shares`, `Owed` and `ConfigAmount` libraries all define a `toUintX` function that unwraps the different types into some `uint`-like type. To be coherent with the use of `unwrap` and all the different `toUint` functions should be consistent unless there's a specific reason to not do it.

Recommendation: Consistently use `toUint`, `toUint16`, etc... Rather than `unwrap` when able.

Euler: Acknowledged, no fix. `toUint` and `unwrap` have a slightly different effects, as the former returns a `uint256` and the latter a smaller type, so they are not interchangable.

Spearbit: Acknowledged.

5.5.47 `interestAccruedFromCache` can avoid extra operations and return earlier

Severity: Informational

Context: EulerSavingsRate.sol#L171-L173

Description: At `interestAccruedFromCache`, when the timestamps are equal (`timePassed = totalDuration`) we would also return `interestLeft`. Therefore, by modifying the same block we will return the same value without doing extra operations.

Recommendation: Consider fast returning when >= as the value will be the same and fewer operations will be performed

```
- block.timestamp > esrSlotCache.interestSmearEnd
+ block.timestamp >= esrSlotCache.interestSmearEnd
```

Euler: Fixed in attached commit 78dfad72.

Spearbit: Verified.

5.5.48 English dialect inconsistencies

Severity: Informational

Context: IRMLinearKink.sol#L43-L60

Description: A common best practice is to use one language and dialect for the sake of consistency, readability and maintainability.

For example: "utilisation" (british) with "s" is used, while in another place "utilize" (american) with "z". Then, in another part is used "initialize" (american), etc...

Recommendation: Consider keeping consistency and only using american or british English.

Euler: Fixed in PR 197.

Spearbit: Verified.

5.5.49 `ESynth` `mint`s are centralized

Severity: Informational

Context: ESynth.sol#L35

Description: `setCapacity` is an admin function that will set the maximum `capacity` of mints an address can have, allowing or disallowing users to mint this way. As the admin is the one who first allows users to mint, this should not be a problem.

However, if for any strange reason someone who initially is set to any non-zero capacity tries to mint, and the admin wants to DoS them by setting it to 0 capacity, they will avoid that person from minting any token, effectively "banning" this person from minting. The logic is as follows:

```
function setCapacity(address minter, uint128 capacity) external onlyOwner {
    minters[minter].capacity = capacity;
    emit MinterCapacitySet(minter, capacity);
}
```

And later checked at `mint` that will revert if not enough capacity is set:

```
if (
    amount > type(uint128).max - minterCache.minted
        || minterCache.capacity < uint256(minterCache.minted) + amount
) {
    revert E_CapacityReached();
}
```

Recommendation: Keep this well documented in the trust model and/or implement a timelock to avoid a sudden change in mint possibilities.

Euler: Acknowledged. Will keep as is.

Spearbit: Acknowledged.

5.5.50 `PegStabilityModule` `swapToUnderlyingGivenIn` and `swapToSynthGivenIn` should early return/revert when `amountOut` is 0

Severity: Informational

Context: PegStabilityModule.sol#L42, PegStabilityModule.sol#L60, PegStabilityModule.sol#L81

Description: Due to rounding down, `amountOut` could be equal to `0` and it could therefore early return or revert to cut unnecessary actions and event emissions.

Recommendation: Add a check to early return or revert when `amountOut == 0`.

Euler: Fixed in attached commit 07683265. Opted to return early when either the `amountIn` or `amountOut` are zero in any of the swap functions.

Opted for returning early to not cause unexpected reverts in potential integrations.

Spearbit: Verified.

5.5.51 Named imports provide more readability

Severity: Informational

Context: LTVUtils.sol#L6

Description: The use of named imports from Solidity files provides clarity and readability. Named imports make it immediately clear which specific functions, contracts, or variables are being utilized from a particular module, reducing ambiguity and making the code easier to understand and maintain.

For example in `LTVUtils`: `import "./types/Types.sol";` should be `import {ConfigAmount} from "./types/Types.sol";`

Recommendation: Use named imports wherever possible, especially for larger modules, to clarify code intentions, simplify maintenance, and improve overall readability.

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5.52 Inconsistent naming decreases the codebase searchability

Severity: Informational

Context: Assets.sol#L14, Core.sol#L15

Description: Consistency is key for a more searchable and maintainable system, either for users, developers, researchers, etc... To know what to look for within the codebase just by a consistent naming, and to understand faster how things should behave.

Some instances of these include:

- Functions like `toUint` be changed to `toUint256`, or their counterparts, `toSharesDownUint256` to `toSharesDownUint`.
- `governor` be changed to `governAdmin`, or `upgradeAdmin` just to `admin`.

Recommendation: Keep naming consistent all over the code.

Euler: Fixed in PR 180. `Core.sol` was removed from scope.

Spearbit: Verified. Euler has decided to use the aliased version `uint` in the nomenclature instead of the full one `uint256`. No changes made to `Governor`.

5.5.53 Helper retriever functions can return dummy data

Severity: Informational

Context: BaseProductLine.sol#L67, GenericFactory.sol#L92

Description: The use of public `push` functions (`createVault`, `createProxy`) with no access control and without their counterpart `pop` function to remove unnecessary elements, can lead to an array full of dummy data that later is iterated in order to create some sort of pagination. These arrays are used as helpers for lens-type contracts.

This shouldn't be a problem in this case as both `start` and `end` items are set. Therefore, under common circumstances there won't be an out-of-gas scenario (maybe in the case of special case `end == type(uint256).max` or in a bad setup). However, if someone wants to fill between real `vaults` or `proxys` values with dummy data, they are able to do so, and external reads would need to filter this data.

Recommendation: Consider adding an admin `pop` function to clean the data and documenting this behavior in the documentation for lens-type contracts

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5.54 Unclear naming can lead to misinterpretation

Severity: Informational

Context: EVCClient.sol#L88-90, Liquidation.sol#L15

Description: Naming is key for understanding different parts of the code with just one look, indeed, wrong naming can lead to wrong assumptions of what the code is going to behave.

Different instances can be improved for a better understanding:

- `hasControllerEnabled` doesn't check if this contract is the account's controller, but if there are *any* controller enabled. It is used correctly but could be renamed to `hasAnyControllerEnabled` to avoid misinterpretation.
- The term `collateralValue` is used ambiguously in `Liquidation.sol` to represent both the unadjusted and adjusted values of collateral. `collateralValue` should **only** be called like that when it represents the value, in units of account, of the full collateral. Consider renaming `collateralValue` to `collateralAdjustedValue` (or similar) when the collateral value (in units of account) is **adjusted** by the LTV.
- `amount` refers to multiple things over the codebase. Sometimes it refers to `shares`, sometimes to `assets` (depending on the context) what leads to a much more less consistent and harder to read codebase (as every instance may be a different thing).

Recommendation: Consider applying some of the aforementioned name corrections.

Euler: Fixed as recommended in PR 179.

Spearbit: Verified.

5.5.55 "*Magic numbers*" should be defined as constants to improve readability and maintainability

Severity: Informational

Context: LiquidityUtils.sol#L119, Cache.sol#L96, ProtocolConfig.sol#L65-L67, IRMSynth.sol#L67

Description: Numbers not defined as constants are less maintainable and readable. Changing them with meaningfully named constants and with proper `@dev` comments, would ease the read, search and maintainability.

- LiquidityUtils.sol#L119: `1e4` should be `CONFIG_SCALE`.
- IRMSynth.sol#L67: `1e18` should be `TARGET_QUOTE`.

Should have a named constant:

- Cache.sol#L96: `1e4 << INTERNAL_DEBT_PRECISION_SHIFT` when `feeAssets` are calculated.

- Cache.sol#L80: 1e27 as input when RPow.rpow is executed.
- Cache.sol#L86: 1e27 when newInterestAccumulator is calculated.

Recommendation: Use named constants for "*magic numbers*" and add comments regarding the expectations of those constants in order to keep a cleaner, more consistent and maintainable codebase.

If those values are shared across the codebase consider moving them to Constants.sol

Euler: We acknowledge the remaining values in Cache, no fix.

Spearbit: Acknowledged.

5.5.56 Unused libraries

Severity: Informational

Context: BalanceUtils.sol#L6, Types.sol#L7, Types.sol#L10, Initialize.sol#L10

Description: Several libraries were left behind after some updates and are not used anymore. Unused code increases the overall complexity of the codebase, making it harder to maintain and read.

Recommendation: Remove unused code in order to improve readability

Euler: Fixed as recommended in PR 177.

Spearbit: Verified.

5.5.57 Event emission can track previous admin role for better monitoring

Severity: Informational

Context: ProtocolConfig.sol#L51, GenericFactory.sol#L41

Description: SetAdmin event logs the new admin for PoolConfig. To improve tracking and monitoring, consider also logging the current admin who initiated the change.

```
function setAdmin(address newAdmin) external onlyAdmin {
    if (newAdmin == address(0)) revert E_InvalidAdmin();

    admin = newAdmin;

    emit SetAdmin(newAdmin);
}
```

The same can be applied to SetUpgradeAdmin event to log the new upgrade admin.

Recommendation: Modify the event emission to include both the new admin and the admin who called the change for better tracking.

Euler: Acknowledged, no fix.

Spearbit: Acknowledged.

5.5.58 `isValidInterestFee` validation can be skipped

Severity: Informational

Context: IProtocolConfig.sol

Description: The documentation should be expanded by explaining that `isValidInterestFee` validation will be skipped by `Governance` if the `interestFee` is updated via `Governance.setInterestFee`, and will be between the bounds (`GUARANTEED_INTEREST_FEE_MIN`, `GUARANTEED_INTEREST_FEE_MAX`).

If that's the case, the boundaries imposed by the `ProtocolConfig` are skipped.

Recommendation: Add proper documentation for this case.

Euler: Acknowledged, no fix. We consider current comments sufficient.

Spearbit: Acknowledged.

5.5.59 `VaultCreated` event can be enhanced for better monitoring

Severity: Informational

Context: BaseProductLine.sol#L74

Description: The `makeNewVaultInternal` function performs an event emission that doesn't fully capture all of the relevant inputs. Consider including all inputs (`upgradeable`, `asset`, `oracle`, `unitOfAccount`) and the value set for `CFG_EVC_COMPATIBLE_ASSET`.

```
function makeNewVaultInternal(bool upgradeable, address asset, address oracle, address unitOfAccount)
↪  returns (IEVault)
{
    // ...
    emit VaultCreated(newVault, asset, upgradeable);
}
```

Recommendation: Enhance the event emission to include all function inputs and `CFG_EVC_COMPATIBLE_ASSET` for better tracking and consistency.

Euler: Acknowledged. Contract was removed from scope.

Spearbit: Acknowledged.

5.5.60 Event emission in `createVault` can be improved

Severity: Informational

Context: Core.sol#L49, Escrow.sol#L28

Description: The function `makeNewVaultInternal` emits an event, which provides valuable tracking information. A similar event could be emitted at `createVault` to track the creation of the vault with its specified parameters.

Recommendation: Implement an event emission to log the creation of the vaults and their parameters for better tracking and consistency with other functions.

Euler: Acknowledged. Contracts were removed from scope.

Spearbit: Acknowledged.

5.5.61 Missing safety checks can lead to undesired behavior

Severity: Informational

Context: Core.sol#L27-L28, Core.sol#L36, BaseProductLine.sol#L51-L52, BaseProductLine.sol#L51-L52

Description: In different contract constructors and some setters, different addresses are set to a variable without checking whether these addresses are non-zero, or if they represent deployed contracts. This contrasts with the approach in other contracts where such safety checks are implemented. Ignoring these checks could lead to some undesired behavior:

- `Core.sol` `constructor` sanity checks:
  - `governor_ !== address(0)`
  - `feeReceiver_ !== address(0)`
- `BaseProductLine.sol` `constructor` sanity checks:
  - `vaultFactory_ !== address(0)`
  - `evc_ !== address(0)`
- `BaseProductLine.sol` `makeNewVaultInternal` fast revert check:
  - `asset !== address(0)` `makeNewVaultInternal` can include a check to fast revert rather than waiting for `isEVCCompatible` to revert.
- `Core.sol` `createVault` safety checks:
  - `oracle!== address(0)`
  - `unitOfAccount!== address(0)`

Recommendation: Consider adding the checks suggested above.

Euler: Acknowledged. Contracts were removed from scope.

Spearbit: Acknowledged.

5.5.62 Missing/wrong comments and typos

Severity: Informational

Context: See each case below

Description: Comments help to provide context and documentation on what different functions, contracts and variables do. Providing clear and precise comments is key to a clean and maintainable codebase. See below a list of related nitpicks:

- Missing comments:
  - GenericFactory.sol#L133 and BaseProductLine.sol#L86 should be documented and explain that the special case where `end == type(uint256).max` exists.
  - BorrowUtils.sol#L66 an inline comment would be useful to explain the logic.
  - AddressUtils.sol#L8 should provide comments regarding the cases in which `checkContract` won't work as expected by the function name due to the check of `code.length` on contracts that, for example, are not yet deployed.
- Unclear comments:
  - AssetTransfers.sol#L25-28 the comment could list and explain with more detail all the possible revert scenarios.
  - Governance.sol#L213 add some extra comments regarding how `origLTV.getLTV(true)` returns the "current" LTV (based on ramping config) and not the target LTV under the ramping mode scenario.
- Wrong comments:

  - Cache.sol#L39-L40 `MarkeStorage` (a misspelled ancestor name) is used instead of `VaultStorage`.
  - Dispatch.sol#L66 the comment states that `no code will run before delegating to module`. However, it does not take into account the `callThroughEVC` modifier that was used previously to `use` modifier in functions like `mint`, `withdraw`, `redeem`, `skim`, `borrow`, `repay`, `loop`, etc...
  - IEVault.sol#L365 includes a stale comment, items in a list can't be duplicated right now.
  - LTVConfig.sol#L44 `timeRemaining < rampDuration` should be `timeRemaining <= rampDuration`.
- Typos:
  - ProtocolConfig.sol#L9 `bech` should be `be`.
  - Constants.sol#L10 `enusure` should be `ensure`.
  - Events.sol#L26 `initiaiting` should be `initiating`.
  - Events.sol#L29 `recipt` should be `receipt`.
  - Events.sol#L36 `receiver` should be `the receiver`.
  - Events.sol#L82-L83 `transfered` should be `transferred`.
  - BalanceForwarder.sol#L12 `a with` should be `with a`.
  - ERCCollateral.sol#L21, Dispatch.sol#L136, RiskManager.sol#L64, RiskManager.sol#L84, VaultStorage.sol#L24 `re-entrancy` should be `reentrancy`.
- Natspec `@return` missing:
  - The `IERC20` Interface is missing the `NatSpec @return`.
- Natspec missing:
  - GenericFactory.sol
  - PegStabilityModule.sol

Recommendation: Improve comments all over the code, correct typos, remove stale comments and fix incorrect comments where possible.

Euler: Fixed in PR 199. Items not addressed in the fix are considered acknowledged.

Spearbit: Verified and acknowledged.

5.5.63 Liquidation Invariants

Severity: Informational

Context: LiquidityUtils.sol#L73-L120, LTVConfig.sol#L33-L49

Description: Euler uses different LTV configurations as well as different prices for liquidation and borrowing calculations. Borrows are accepted when `healthScore(borrow) > 1.0`, and liquidations are performed when `healthScore(liquidation) <= 1.0` with `healthScore(x) := collateralValue(x) * getLTV(x) / liabilityValue(x)`. It's important that an accepted health check when borrowing does not immediately lead to an unhealthy position regarding liquidation checks. We can prove this by showing `healthScore(liquidation) >= healthScore(borrow)`.

- `getLTV` **invariant:** The following holds for `getLTV`:

  ```
  getLTV(borrowing) <= getLTV(liquidation)
  ```

  Proof: From the code we can distinguish the cases:

  1. `targetLTV >= originalLTV`: `getLTV(borrowing) = getLTV(liquidation) = targetLTV`.
  2. `targetLTV < originalLTV`: `getLTV(borrowing) = targetLTV <= lerp(originalLTV, targetLTV) = getLTV(liquidation)`.

- **Health invariant:** The following holds for `healthScore(x)`:

```
healthScore(borrow) <= healthScore(liquidation)
```

Proof:

```
collateralValue(borrow) * getLTV(borrow) / liabilityValue(borrow)
<= collateralValue(liquidation) * getLTV(liquidation) / liabilityValue(liquidation)
```

This follows from:

1. `collateralValue(borrow) <= collateralValue(liquidation)` as borrow uses bid prices compared to liquidations using the mid price.
2. `liabilityValue(borrow) >= liabilityValue(liquidation)` as borrow uses ask prices compared to liquidations using the mid price.
3. `getLTV(borrowing) <= getLTV(liquidation)` by the `getLTV` invariant.

Recommendation: The oracles need to guarantee that the mid-price quote used for liquidations (`oracle.getQuote()`) is indeed in between the (bid, ask) quotes used for the borrows (`oracle.getQuotes()`). The `oracle.getQuotes()` function must also guarantee `bid <= ask`. Document these assumptions on the oracles.

Euler: Acknowledged. We updated the white paper with these assumptions.

Spearbit: Acknowledged.