Electisec

October 14, 2025

Prepared for
Euler Finance

Audited by
HHK
engn33r

Euler eUSD And Crosschain Fee Flow Review

Smart Contract Security Assessment

Contents

Review Summary

1 Protocol Overview

eUSD is a synthetic cross-chain stablecoin that will use Euler's EVK vault structure to direct yield from borrowing fees on all chains to a single vault on Ethereum mainnet.

2 Audit Scope

This audit covers 7 smart contracts totaling approximately 537 lines of code across 2 days of review.

```
src/
├── ERC20
│   ├── deployed
│   │   └── ERC20Synth.sol
├── FeeFlow
│   └── FeeFlowControllerEVK.sol
├── IRM
│   └── IRMBasePremium.sol
├── OFT
│   ├── OFTFeeCollector.sol
│   └── OFTGulper.sol
└── Util
    ├── FeeCollectorGulper.sol
    └── FeeCollectorUtil.sol
```

3 Risk Assessment Framework

3.1 Severity Classification

4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	4
Informational	6

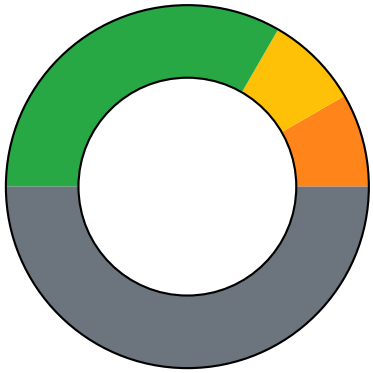

Figure 1: Distribution of security findings by impact level

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

5 Overall Assessment

Audit Overview

1 Project Information

Protocol Name: Euler Finance
Repository: https://github.com/euler-xyz/evk-periphery
Commit Hash: 642602f8cb487969932d86298993f40b42727c88
Commit URL: https://github.com/euler-xyz/evk-periphery/commit/642602f8cb487969932d86298993f40b42727c88

2 Audit Team

HHK, engn33r

3 Audit Resources

Code repository and documentation

Category	Mark	Description
Access Control	Good	Proper access control is implemented for all sensitive functions, with some room for minor improvements.
Mathematics	Excellent	No complex mathematics involved, and all operations are safe.
Complexity	Good	The contracts are straightforward and well-organized, though the cross-chain component adds some complexity.
Libraries	Excellent	Uses OpenZeppelin and LayerZero libraries, both of which are battle-tested.
Decentralization	Low	The protocol has strong control over eUSD supply and peg stability. The cross-chain fee transfer depends on LayerZero infrastructure stability and Euler configuration.
Code Stability	Good	The code remained stable during the review, though the tests were still a work in progress.
Documentation	Excellent	Great documentation, diagrams, and NatSpec provided.
Monitoring	Good	Events are emitted throughout the lifecycle, although a couple of functions could benefit from custom events.
Testing and verification	Low	Very low to no testing at the time of review, as tests were still being written.

Table 1: Code Evaluation Matrix

4 Critical Findings

None.

5 High Findings

5.1 Layer Zero dust removal causes fee harvesting DOS

The Layer Zero dust removal mechanism can cause slippage checks to revert, resulting in DOS of cross-chain fee harvesting.

Technical Details

Layer Zero's `send()` function allows the sender to specify `amountLD` and `minAmountLD`. If `minAmountLD` exceeds `amountLD`, the transaction reverts to protect users from unexpected fees.
When transferring cross-chain, Layer Zero converts amounts from 18 decimals to 6 decimals and removes dust from the transfer in the `_removeDust()` function. It ensures the rounded-down amount is still greater than `minAmountLD`, reverting otherwise.
In `collectFees()`, both `amountLD` and `minAmountLD` are set to the token balance available after calling `_convertAndRedeemFees()`.
In `buy()`, both values are set to the `paymentAmount` determined by the Dutch auction.
In both cases, the contract determines the value rather than the caller. For 18-decimal tokens, the amount may have values beyond six decimals. Layer Zero's dust removal will reduce `amountLD`, causing it to no longer be greater than `minAmountLD`, resulting in a revert.
This makes fee harvesting difficult to execute, as it will often revert when the rounded values don't align or if an attacker sends dust to trigger reverts. This causes temporary DOS of fee harvesting and cross-chain transfers.

Impact

Medium. Fee harvesting and cross-chain transfers can be DOS.

Recommendation

In `buy()`, remove dust from `paymentAmount` before transferring from the caller.
In `collectFees()`, remove dust from the balance before passing it to Layer Zero and leave the dust in the contract for future calls.

Developer Response

Fixed in commits e47565f407103a1961491eef4529ebe56b0fe047 & 130e3df2c24f0be2cdbfe154aabd45f058a5fd43 & faf562ef992882143c467eaf30def06ed7daf9f7.

6 Medium Findings

6.1 Contracts cannot receive ETH for Layer Zero fees

The `FeeFlowControllerEVK` and `OFTFeeCollector` contracts lack a mechanism to receive ETH needed to pay for Layer Zero cross-chain transfers.

Technical Details

Both `FeeFlowControllerEVK` and `OFTFeeCollector` use Layer Zero to transfer fees cross-chain by calling `IOFT.send{value: fee.nativeFee}(params)`, which requires the contracts to hold ETH. However, the functions calling this transfer are not `payable`. Neither contract implements a `receive()` or `fallback()` function, making it impossible to send ETH to them. This will cause harvesting transactions to revert due to insufficient ETH for Layer Zero fees. The inability to receive ETH will also prevent any refund of excess fees.

Impact

Low. The contracts will be unusable for cross-chain fee transfers.

Recommendation

Add a `receive()` function to `OFTFeeCollector` contract and make `buy()` payable on the `FeeFlowControllerEVK` to allow them to receive ETH for Layer Zero fees.

Developer Response

Fixed in commit 471ca2de2d5804a66a47b2a90620835a065dcb86.

7 Low Findings

7.1 Missing EVC compatibility in FeeCollectorUtil access control

The `FeeCollectorUtil` contract does not override `AccessControlEnumerable` functions to be compatible with the EVC.

Technical Details

Other contracts inheriting `AccessControlEnumerable`, such as `ERC20Synth` and `IRMBasePremium`, override access control functions to be compatible with the EVC by adding `EVCUtil`'s `_msgSender()` and `onlyEVCAccountOwner()`. However, `FeeCollectorUtil` lacks these overrides.

Impact

Low. Access control functions are not callable through the EVC.

Recommendation

Update the contract to override access control functions with EVC compatibility.

Developer Response

Fixed in commit 60dfa7bc1742b914b1527bd1f1be25eff3be468a.

7.2 Immediate interest rate change possible in IRMBasePremium

Technical Details

The IRMBasePremium.sol interest rate model does something that no other common Euler IRM does: allow immediate changes to interest rates from a centralized actor. Any address with the `RATE_ADMIN_ROLE` role can modify the interest rate charged to borrowers with immediate effect.
A malicious address with such `RATE_ADMIN_ROLE` permissions could wait until large depositor addresses into the vault have the least on-chain activity, then boost and call `setBaseRate()` and `setPremiumRate()` to set maximum interest rates. This would quickly drain all value from the vault's borrowers, causing liquidations and loss of value.

Impact

Low. The IRM contract is highly centralized, and a malicious `RATE_ADMIN_ROLE` could negatively impact borrowers without warning.

Recommendation

This contract is missing hardcoded protection for borrowers. Some options for borrower protection measures include: - A timelock delay on modifying rates. This could be implemented in the multisig that receives the `RATE_ADMIN_ROLE` role - An `ADJUST_INTERVAL` value, like what is implemented in IRMSynth.sol, to avoid changes in interest rate having an immediate effect on borrowers - A maximum value for `baseRate` and `premiumRate` so that a guaranteed maximum borrowing rate is enforced. This exists in other Euler IRM contracts in the IRM factory as the constant `MAX_ALLOWED_INTEREST_RATE`, which has a limit of 1000% APY. But other IRM contracts are immutable and can use this approach, while `IRMBasePremium.sol` is not immutable and cannot rely on limits set in a factory.

Developer Response

Fixed in commit 5d817390e339962aed5eefddb666037aaaa37fa7.

7.3 Role misalignment for `_ignoredForTotalSupply` modifications

Technical Details

In ERC20Synth.sol, two functions can add an account to `_ignoredForTotalSupply`. `allocate()` has a modifier to only allow `ALLOCATOR_ROLE` to call it. At the same time, `addIgnoredForTotalSupply()` has a modifier to only allow `DEFAULT_ADMIN_ROLE` to call it. Enforcing `DEFAULT_ADMIN_ROLE` on `addIgnoredForTotalSupply` gives a false sense of security, because `allocate()` performs the same action but allows roles other than the admin to call it.
Note that this was not an issue with the older ESynth.sol because there were no separated roles in ESynth.sol. There was just a single `onlyOwner` access control modifier on all such functions.

Impact

Low. Access control for key actions is inconsistent.

Recommendation

Modify the access control on `addIgnoredForTotalSupply()` to `onlyRole(ALLOCATOR_ROLE)` to align it with `allocate()`.
An alternative approach that can save gas is to remove `_ignoredForTotalSupply.add(vault);` from `allocate()` entirely, perhaps adding a `require(_ignoredForTotalSupply.contains(vault))` check as a replacement (which might also be good to include in `deallocate()`). This would rely on the `DEFAULT_ADMIN_ROLE` to add vaults before later allowing `ALLOCATOR_ROLE` to `allocate()` and `deallocate()` to those specific vaults.

Developer Response

Fixed in commit c4ee48d9396469728cbe8070b7544ce098747e3b.

7.4 Toggling infinite minting doesn't reset old `minted` values

Technical Details

If a minter address initial has a finite minting capacity, then receives an infinite minting capacity, and then is returns to a finite minting capacity again, the old `minters[minter].minted` value will not have increased during the period when the minter had an infinite minting capacity. At the same time, the `minters[minter].minted` value will still have decreased due to burn operations calling `_decreaseMinted()` when an infinite minting capacity was set. The storing of the old `minters[minter].minted` value when the minter is modified to have infinite minting capacity may not be intended behavior.
This was not relevant in the old ESynth.sol logic, because there was no infinite minting and incrementing `minterCache.minted` was always performed. But in the newer ERC20Synth.sol logic, `minterCache.minted` is not modified when `minterCache.capacity == type(uint128).max`.

Impact

Low. Irrelevant or unused data can remain in the contract and could alter the logic flow of the code in the future.

Recommendation

To clear the old `minters[minter].minted` value when a minter receives an infinite minting capacity, add this line to the `setCapacity()` function. This will also improve the efficiency of `_decreaseMinted()` calls with this minter.

```
function setCapacity(address minter, uint128 capacity) external
onlyEVCAccountOwner onlyRole(DEFAULT_ADMIN_ROLE) {
    _grantRole(MINTER_ROLE, minter);
    minters[minter].capacity = capacity;
+   if (capacity == type(uint128).max) minters[minter].minted = 0;
    emit MinterCapacitySet(minter, capacity);
}
```

Developer Response

Fixed in commit 1ecb2ec0d0e6664e065278c429be32b44243b241.

8 Gas Savings Findings

8.1 Gas efficiency improvement in ERC20Synth mint()

Technical Details

The if statement logic in ERC20Synth.sol `mint()` can be made more efficient.
The logic currently looks like this:

```
if (amount > type(uint128).max - minterCache.minted
    || minterCache.capacity < uint256(minterCache.minted) + amount)
```

The uint256 casting can be removed by rearranging the 2nd inequality.

```
if (amount > type(uint128).max - minterCache.minted
    || amount > minterCache.capacity - minterCache.minted)
```

And now we can see that the 2nd inequality is always more strict than the first one, because `capacity` is a uint128 and is not equal to type(uint128).max at this point in the logic. Since `minterCache.capacity` < `type(uint128).max`, the first check is redundant. We now have

```
if (amount > minterCache.capacity - minterCache.minted)
```

To guarantee there is no underflow, an additional check must be added.

```
if (minterCache.capacity < minterCache.minted) revert CapacityReached();
if (amount > minterCache.capacity - minterCache.minted) revert CapacityReached();
```

This removes a casting operation and an addition compared to the original logic while maintaining the same result.

Impact

Gas savings.

Recommendation

Make the recommended change.

Developer Response

Fixed in commit 6c42feb4a01b8f6b34e57a41ae1f6891efa922ed.

9 Informational Findings

9.1 Missing events in ERC20Synth

Technical Details

The following functions in `ERC20Synth` could emit custom events for better indexing: `allocate()`, `deallocate()`, `addIgnoredForTotalSupply()`, and `removeIgnoredForTotalSupply()`.

Impact

Informational.

Recommendation

Add events to these functions to improve indexing.

Developer Response

Fixed in commit fe2b52a6e3761d6a5322474f585ef6b2e56fb39b.

9.2 Missing assertion in FeeCollectorGulper constructor

Technical Details

FeeCollectorGulper.sol has a similar role to OFTGulper.sol, both send funds to the ESR vault. But in the constructor of OFTGulper, the `feeToken` value is confirmed to be `esr.asset()`, while no such confirmation or assertion exists in the constructor of FeeCollectorGulper.sol. This check is not mandatory, but it provides greater alignment between the gulper contracts and greater confidence that all values are correctly set on deployment.

Impact

Informational.

Recommendation

Add a line with logic like this to the constructor of FeeCollectorGulper.sol.

```
require(_feeToken == EulerSavingsRate(_esr).asset());
```

Developer Response

Fixed in commit 4defa3247a93dd327a3d8a8af6613dcfdd4c1ab9.

9.3 Imports can be combined

Technical Details

Some solidity imports can be combined.
In FeeCollectorGulper.sol:

```
- import {IERC20, SafeERC20} from "openzeppelin-contracts/token/ERC20/utils/
SafeERC20.sol";
- import {FeeCollectorUtil} from "./FeeCollectorUtil.sol";
+ import {FeeCollectorUtil, IERC20, SafeERC20} from "./FeeCollectorUtil.sol";
```

In ERC20Synth.sol:

```
- import {Context} from "openzeppelin-contracts/utils/Context.sol";
- import {AccessControl} from "openzeppelin-contracts/access/AccessControl.sol";

- import {IAccessControl} from "openzeppelin-contracts/access/IAccessControl.
sol";
+ import {AccessControl, IAccessControl, Context} from "openzeppelin-contracts/
access/AccessControl.sol;
```

Impact

Informational.

Recommendation

Consider combining imports to reduce the number of separate files referenced and to reduce the SLOC count.

Developer Response

Fixed in commit 9e25f4773e5881b74539aaa55fd70060aa002fbc.

9.4 Use consistent `recoverToken` logic

Technical Details

OFTGulper.sol and FeeCollectorUtil.sol implement a `recoverToken()` function. The `recoverToken()` implementation in FeeCollectorUtil.sol supports native tokens, but OFTGulper.sol does not. This inconsistency is illogical, because OFTGulper.sol has a payable function `lzCompose()` that could receive native tokens, but it cannot recover them. FeeCollectorUtil.sol does not have a payable function or any other method to receive native tokens, but it does support the recovery of native tokens.

Impact

Informational.

Recommendation

Implement `recoverToken()` the same way to support native token recovery in OFTGulper.sol and FeeCollectorUtil.sol. This requires modifying the implementation in OFTGulper.sol.

Developer Response

Fixed in commit fb346bb1176ad884f2ecff5ba8c1c1ed5625ef53.

9.5 NatSpec typos

Technical Details

There are several typos in the NatSpec comments of the code. - In ERC20Synth.sol, the NatSpec comment for `renounceRole()` includes the text ”must match msg.sender”, but since EVCUtil is inherited, the text should say ”must match _msgSender()” to explain there is support for calls via the EVC. - The same typo exists in IRMBasePremium.sol for `renounceRole()`. - In ERC20Synth.sol, the NatSpec for the constructor says ”Deploys the ESynth contract.” instead of ”Deploys the ERC20Synth contract.” - In FeeFlowControllerEVK.sol, ”Continous” should be spelled ”Continuous”. - The `buy()` function of FeeFlowControllerEVK.sol has a comment about the hook stating ”We do not check the success of the call as we allow it silently fail”, when it should say ”we allow it to silently fail”.

Impact

Informational.

Recommendation

Fix typos as explained above.

Developer Response

Fixed in commit 016c883b4fea3ee204087fa60e8b3dcc63ed6148.

9.6 AccessControlDefaultAdminRules can offer additional security

Technical Details

Since 4.9.0, OpenZeppelin offers AccessControlDefaultAdminRules.sol, which adds additional security measures that may be useful for a high-security contract, such as a governance-managed stablecoin. Some of the security benefits include: - Only one account holds the `DEFAULT_ADMIN_ROLE` - Enforces a 2-step process to transfer the `DEFAULT_ADMIN_ROLE` - Enforces a configurable delay between the two steps

Impact

Informational.

Recommendation

Consider introducing AccessControlDefaultAdminRules.sol to ERC20Synth.sol to add additional security around role management.

Developer Response

Acknowledged. Will keep as is for behavior consistency with other access-controlled smart contracts in the system.

10 Final Remarks

The eUSD contracts are straightforward, leveraging already audited contracts such as ERC20Synth and FeeFlowController with a few modifications. The cross-chain aspect through LayerZero adds some complexity, which requires proper testing. Unfortunately, at the time of review, very little to no testing was provided, resulting in some findings that could have been avoided. We are confident that Euler will complete proper testing before deployment.
Some concerns were raised around the peg stability and architecture of eUSD. While the contracts seem sound, the big picture remains unclear. This review focuses solely on the cross-chain fee transfer and synth contracts, and not on the safety of their usage within the broader system.
The Euler team provided extra fixes not directly linked to findings on this report that were also reviewed inside PR#368, up to commit 4711a8ad0f23bdc0209b7be772efa42b887e3518.