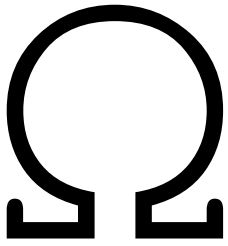
Euler Fee Flow

Final Audit Report

February 27, 2024



Team Omega

Teamomega.eth.limo

Summary

Euler Finance has asked Team Omega to audit the contracts that define the behavior of their fee flow controller system.

We found **no high or medium severity issues**. We classified **4** issues as “low”, and an additional **3** issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Severity	Number of issues	Number of resolved issues
High	0	0
Medium	0	0
Low	4	4
Info	3	3

On the whole, we found the code is well-documented, well-tested and well-written.

Scope of the Audit

The audit concerns the code developed in the following repository:

```
https://github.com/euler-xyz/fee-flow.git
```

And in particular, the Solidity files in the `src` directory in that repository.

The audit report was based on the following commit from February 9, 2024

```
0b67c0308a80835c8aa4a5d7011ed349bcbb8445
```

Resolution

The team at Euler Finance addressed the issues in commit

```
03b445222d31f60741d785c10696ce030a7fdf87
```

We have checked the fixes and updated the issues below

Methods Used

The contracts were compiled, deployed, and tested in a test environment. Two auditors independently reviewed the code - the current report is the result of combining our findings.

Code Review

We manually inspected the source code to identify potential security flaws.

Automatic analysis

We have used static analysis tools to detect common potential vulnerabilities. No high severity issues were identified with the automated processes. Some low severity issues were found, and we have included them below in the appropriate parts of the report.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

About Team Omega

Team Omega (teamomega.eth.limo) is a small team that consists of Jelle Gerbrandy and Ben Kaufman. They have been working on blockchain projects, and Solidity in particular, since 2017, and have been doing smart contract audits since 2021.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion

Description

The software implements a Dutch Auction, in which fees from markets are accrued and sold. Roughly, there are two on-chain actions that are relevant here:

- Anyone can claim fees from a market, which will flow to the auction contract
- Anyone can buy the tokens from the auction contract for the currently published price

As there are no incentives for other agents other than the buyer to claim the fees, we can assume claiming the fees from the markets and buying the tokens in the auction will happen in a single transaction - i.e that the buyer not only buys the tokens in the contract, but will also claim the tokens and pay the corresponding fees in the same transaction.

The contract does not just function as a Dutch auction to sell a given batch of tokens, as buyers also claim the fees, and decide when and where to do that.

The logic here is more complex than a classical Dutch auction, because (a) the amount and value of the goods for sale may change during the auction and (b) the buyer must pay for gas costs, which also vary over time.

More formally, the contract has a number of immutable variables that define the auction behavior:

`epochPeriod`: the (maximum) length that an auction will run
`minInitPrice`: the minimum starting price of an auction

The price function then is given by:

```
price(t) = initPrice * (1 - (t-startTime)/epochPeriod))
```
(for t <= startTime + epochPeriod, it is 0 otherwise)

Other values that influence the outcome of the auction are:

`gas(t)`: the amount of gas the buyer pays - depends on the gas price, the amount of tokens claimed, the path the buyer uses to sell the tokens, etc.
`value(t)`: the total value of the tokens that can be claimed - depends on which fees are being claimed, and the market price of the claimed tokens

If there are enough buyers competing, then the auction will settle at a time t such that:

`price(t)= value(t) - gas(t) - δ` (where δ is the profit for the buyer)

(We have simplified somewhat here - the `value` function is not the same for all buyers, as buyers can choose which tokens to claim, and may have access to markets that other buyers do not have; similar considerations hold for the `gas` function.

As buyers compete to optimize for δ, they will naturally seek the best `value` (i.e. get the best price for the tokens) and minimize `gas` (i.e. make their transactions as gas efficient as possible, and seek out lower market prices for gas).

The protocol instead is looking to get the best trade for their tokens, i.e the protocol would like to get a price that is as close as possible to the actual value of the tokens, and so would optimize for `price(t)/value(t)`, or, in other words, have relatively as little `gas(t)` as possible in proportion to the value of the tokens, and keep δ as low as possible.

Buyers, however, have no incentives to optimize for the proportion of gas relative to the value. This discrepancy in incentives between protocol and buyers may lead to suboptimal outcomes for the protocol when fees accumulate relatively slowly, as described in F1.

Besides the gas costs, the second factor that influences whether the result is optimal for the protocol is the value of δ - i.e. the profit for the buyer.
We can probably assume that changes in gas costs and the prices of the underlying tokens are continuous (meaning there are no sudden jumps) - or at least close enough to make any inefficiencies acceptable. If this is not the case, for example, if there is a sudden increase in the token price just when the auction is about to settle, the auction may settle at a price that is significantly below the market price. This is probably an acceptable inefficiency.

If the `gas` and `value` functions are continuous (i.e. there are no sudden jumps), and if there are enough competitors, then the competition between the buyers will make the value of δ approximate 0. However, the total `value` of the tokens being auctioned does not just depend on their market price, but also on the schedule according to which the fees become available, and the latter may not be continuous at all. This is the underlying issue in F2 below.

Findings

General

G1. Pin versions of solidity, and of solidity dependencies [low] [resolved]

The file `.gitmodules` specifies some smart contract dependencies. Of these dependencies, solmate and ethereum-vault-connector. These dependencies are specified as generic links to the source repositories:

```
[submodule "lib/solmate"]
	path = lib/solmate
	url = https://github.com/transmissions11/solmate
[submodule "lib/ethereum-vault-connector"]
	path = lib/ethereum-vault-connector
	url = https://github.com/euler-xyz/ethereum-vault-connector
```

If developers of these dependencies push a new commit to the default branch of these repositories, then, on the next fresh install, or when a developer updates the dependencies, the compiled code will change, which can cause ambiguity as to which specific version of these repositories was used to generate the bytecode once the contract is deployed.
This could raise issues later on when trying to recompile the contracts, for example for verification on Etherscan, as it is possible that a new version will be used and cause the bytecode to change and not match the deployed bytecode.

It also represents a small attack surface, in which owners of one of these repositories could push malicious changes just before deployment, which would then be merged into the code.

Similar considerations hold for the specification of the Solidity versions. Even if the `foundry.toml` is unambiguous about the version of Solidity to use (namely 0.8.20), `FeeFlowController.sol` specifies the solidity version as a range:

```
pragma solidity ^0.8.20;
```

Recommendation: Specify fixed versions of smart contract dependencies in `.gitmodules`. This can be done by referring to specific commits of the software like so:

https://github.com/transmissions11/solmate@c892309933b25c03d32b1b0d674df7ae292ba925

Also, pin a specific solidity version:

```
pragma solidity 0.8.20;
```

Severity: Low
Resolution: The issue was resolved as recommended.

G2. No License information [low] [resolved]

The repository has no license information, and the solidity files are marked as:

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

The main dependencies of the software, `solmate` and `ethereum-vault-connector`, are published under the GPL, which states that:

```
b) You must cause any work that you distribute or publish, that in
whole or in part contains or is derived from the Program or any
part thereof, to be licensed as a whole at no charge to all third
parties under the terms of this License.
```

Recommendation: Although the terms of the GPL are not completely unambiguous (and we do not give legal advice), if there is no reason to do otherwise, publish this code under the terms of the GPL.
Severity: Low
Resolution:

G3. Certora formal verification is broken and lacks invariants [info] [resolved]

The project uses Certora to perform formal verification for the code. However, formal verification specs seem to be out of date, and do not work with the current version of the code.

Additionally, the formal verification could benefit from adding additional invariants. Here are some potential invariants that could be added to the formal verification process:

- Balance of fee flow controller of a bought asset is always 0 after buy
- Balance of asset receiver is incremented by balance of fee flow controller after buy
- Balance of buyer is reduced by payment amount, and never by more than max payment amount
- Balance of payment receiver is increased by payment amount
- Payment amount returned on buy is never higher than maximum payout
- Epoch Id is always incremented by 1 after buy (and becomes 0 if it reaches max uint16)

Recommendation: Fix the formal verification and add the recommended invariants. Also, you should consider adding the formal verification to your continuous integration process, so you are sure that the verification will run after you make any new changes.
Severity: Info
Resolution: The issue was resolved as recommended. Formal verification now passes and all recommended invariants were added.

FeeFlowController

F1. Auction is not always gas efficient [low] [acknowledged]

There are cases in which the auction can be inefficient because a large part of the bid is paid in gas costs. Consider the following example:

```
epochTime: 100 time units
fees claimable during epoch: $100
Gas costs for claiming, transferring and selling: $50
minInitPrice: $1000
Initial price: $1000
```

If we start this auction, it will settle around 96 time units into the auctions, at a price of $40. The winner will claim the $96 worth of fees and take them from the contract, pay $50 in gas, and have a profit of $6. If the parameters do not change the auction will now restart with the same parameters, and settle with the same price, so the system is in a (very precarious) equilibrium.
This is highly inefficient, as the auction will basically return less than 50% of the value of the fees to the protocol.
It is easy to find other examples. For example, if the fees claimable during `epochTime` are just slightly above the gas required to claim these fees, the auction will settle at a price close to $0 each time.

Recommendation: The problem in this example is that fees are claimed very often, and so a disproportionate amount of money is paid for gas.
Care should be taken to choose the `epochTime` and a `minInitPrice` in such a way that the amount of fees accumulated when the auction settles is much higher than the expected gas costs that a competitor will typically pay.
Another option would be to introduce a reserve price for the auction, instead of driving the price all the way down to 0. Choose a value for the reserve price that is (significantly) over the expected gas price.
Resolution: The developers acknowledged the issue, and will consider parameters before deploying.

F2. Fee streams must accumulate continuously for the auction to be optimal [low] [acknowledged]

The auction may sell tokens under their market price if fees do not come as a continuous stream, but rather as discrete steps. Specifically, if an auction is close to its (current) settlement price, and then a market generates a large amount of fees, these extra fees will basically be donated to the buyer.
For a specific example, suppose an auction is at price $100 and has accumulated $100 of fees that are claimable for $1 in gas costs - i.e. the auction is very close to settling. At the next tick, one of the markets generates $100 in fees. A buyer can then claim these fees, and buy $200 worth of tokens for $100.

Note that this fact can be abused by an attacker to avoid paying fees in one of the markets. Suppose, for example, that a user withdraws $1M in a market maker that has a 1% withdrawal fee. They can wait for the auction to almost settle, and then, in a single transaction, withdraw their collateral, claim the $10,000 of fees to the auction contract, and buy back their fees basically for free from the auction contract.
Recommendation: Make sure that you are integrating only fee streams that generate a more or less continuous stream of fees (or be prepared for the resulting losses).
Severity: Low
Resolution: The developers acknowledged the issue, and will ensure to use the contracts in the appropriate context.

F3. Declare functions as external instead of public when possible [info] [resolved]

The `getPrice` and `getSlot0` functions are marked as `public`, but are not used within the contract and can be declared `external`.
Recommendation: Mark `getPrice` and `getSlot0` as `external`.
Severity: Info
Resolution: The issue was resolved as recommended.

F4. Initial start time should be configurable [info] [acknowledged]

The contract is deployed with the start time for the first auction being set to `block.timestamp`, which means the auctioning process starts immediately upon deployment. Yet it is likely that, when deployed, not only will it not have any assets to sell, but it will not even be configured as the fees recipient anywhere. Meanwhile, the price immediately starts to drop while auctioning should not have started yet.
Recommendation: Allow specifying the initial start time for the first auction in the constructor.
Severity: Info

Resolution: Acknowledged. The project has chosen to favor contract simplicity over getting the optimum price in the first epoch.