Euler Finance

Audit Report

Prepared by Christoph Michel

December 9, 2021.

Contents

1 Introduction

Euler Finance is a novel permissionless lending protocol on Ethereum. The documentation can be found at docs.euler.finance.

The auditors were tasked by Sherlock to review the codebase as part of their insurance assessment of Euler Finance.

1.1 Scope of Work

The auditors were provided with a GitHub repository at commit hash 187d6f7. There has **not** been a *code freeze* for this commit, the contracts are still in development, the team indicated that these contracts are close to the contracts which will be used for the actual deployment.

The task was to audit the contracts, consisting of the following files with their sha1 hashes:

File	SHA1
BaseModule.sol	4319077121f33142b37b84046a8acb6d591c6e99
Events.sol	48a78f34868b495a42fba51f993deade4ded924e
Euler.sol	9f0ebe3c61916bae36fc25a1ca9aedcd73876a8d
BaseLogic.sol	6b6df0b546afa8be60a61be102d65241a9bb8203
BaseIRMLinearKink.sol	a27c0e982949f7518b697282fd6947bc938a2472
PToken.sol	2d58690a7dfc693140c7f1451c6e2e55d6e225ea
Proxy.sol	f5ab69fc4fa8c8c3ef403426d7ff029f92a17a4a
Storage.sol	0b23f3859d8e32c4ce4387d017ff5197bd9fdcf5
adaptors/FlashLoan.sol	4fbf9a1cd50ae75b97528f63616adeb30879dae7
Base.sol	d7ccc78c66f3e031698676418844d6a4f4ad9c84
Constants.sol	8683706370a0249f759fe2be68c9d2bc041ae49a
BaseIRM.sol	14123d758060cb242f225d2f8379605a7ac0caf4

File	SHA1
modules/Markets.sol	71ec9704cfbc3d175e902199ac3c3191626a0761
modules/EToken.sol	573909c7dcd26d69ace240c2a6f7132b47d2cd19
modules/DToken.sol	71970d845222af4ed0f93e1c8d56755472aeb44f
modules/Governance.sol	49807bfa9c8c0ba81a8895e5b3f83748d3aa1747
modules/Exec.sol	a7564f71b48b60295b3b328a69b0b2dd547f3632
modules/RiskManager.sol	d89aa22a264a6194f8936a50325d452f9185831b
modules/interest-rate-models/IRMDefault.sol	9ee1401957ef90e2a8978756d2e9b10639871ee5
modules/interest-rate-models/test/IRMZero.sol	e1fe2973bea9bafb9832eb82fcdc2c1a64121c31
modules/interest-rate-models/test/IRMFixed.sol	d58189d31c30c0e2ed5b0f7f7c684c738f1e346b
modules/interest-rate-models/test/IRMLinear.sol	027ce717868b320a2ae3923503c05b56951b91a0
modules/interest-rate-models/IRMClassStable.sol	18ff58e77d2d0c598b7f7111801685e4b72f5e88
modules/interest-rate-models/IRMClassMajor.sol	67629d6a22f38be782979d465cee6851cd54dfc1
modules/interest-rate-models/IRMClassMidCap.sol	0414db2a5043d9de7f5c57f722f6245677f675ee
modules/Swap.sol	dbb293092456c5c1f68a61b0f9005e9726e1abc2
modules/Installer.sol	4dddd03c5f8f0da246194a1dd0732a9de79e237e
modules/Liquidation.sol	a550910b7e4739859c580263af93c6dd0a5fd598
views/EulerGeneralView.sol	18c7b483432585c19fbca7c04f095eb993cb4b34
Utils.sol	6b233eb7737baa43e6f9e55172fdfb29d762386e
IRiskManager.sol	e49619e37a291bd3f1972b272f4b1e267b3d5883
vendor/IUniswapV3SwapCallback.sol	f13bf3c597f26d70800687a222d026b9b7a928eb
vendor/TickMath.sol	9d3814b833f02b56a4dcd33b8f2f08604bc20c88
vendor/ISwapRouter.sol	9f67753ed6044f3617523b1db4d740e7ef265b7f
vendor/RPow.sol	be96bf7a014179a4410e2c7d28e0e5c92bc4a37a
vendor/FullMath.sol	9a5c7d6523f3546c42f24296846f0762c127ed84
Interfaces.sol	d81870507ee0cdb7b7bb45ef44fa78ad3908b690

The rest of the repository was out of the scope of the audit.

1.2 Security Assessment Methodology

The smart contract's code is scanned both manually and automatically for known vulnerabilities and logic errors that can lead to potential security threats. The conformity of requirements (e.g., specifications, documentation, White Paper) is reviewed as well on a consistent basis.

1.3 Auditors

Christoph Michel

2 Severity Levels

We assign a risk score to the severity of a vulnerability or security issue. For this purpose, we use 4 *severity levels* namely:

MINOR

Minor issues are generally subjective in nature or potentially associated with topics like “best practices” or “readability”. As a rule, minor issues do not indicate an actual problem or bug in the code. The maintainers should use their own judgment as to whether addressing these issues will improve the codebase.

LOW

Low-severity issues are generally objective in nature but do not represent any actual bugs or security problems. These issues should be addressed unless there is a clear reason not to.

MEDIUM

Medium-severity issues are bugs or vulnerabilities. These issues may not be directly exploitable or may require certain conditions in order to be exploited. If unaddressed, these issues are likely to cause problems with the operation of the contract or lead to situations that make the system exploitable.

HIGH

High-severity issues are directly exploitable bugs or security vulnerabilities. If unaddressed, these issues are likely or guaranteed to cause major problems or, ultimately, a full failure in the operations of the contract.

3 Discovered issues

3.1 Uniswap V3 oracle can be manipulated without much risk (high)

Euler Finance uses Uniswap V3's TWAP oracle to price its assets. This price is used to calculate the collateral and liability values.

The Uniswap V3 prices are generally easier to manipulate than Uniswap V2 because of the *concentrated liquidity* feature that makes sure that most liquidity is provided at a small range around the "real price" (external market price). Because of this, an attacker can burn through this range order with an average execution price close to the real price. The attacker does not lose much value on the trade due to the low slippage. (Although it requires the attacker to own the capital as it cannot be flashloaned due to having to maintain the new price across one block.)

They can then create a tiny liquidity position at the highest tick (which correlates to a price close to infinity), which will be the tick the pool currently trades at. If this position is not arbitraged within the same block, the high tick will be added to the oracle's `tickCumulatives` value on the next block and drastically increase the TWAP.

The attacker could perform this price manipulation for a collateral asset which is then valued at an inflated price and use it to borrow all Euler assets (simply through cross-borrowing or borrowing isolated assets by creating new subaccounts with the collateral asset.)

Recommendation

We recommend using a second external oracle for all collateral assets with a TWAP which is harder to manipulate. Governance needs to ensure that such an oracle exists before listing an asset as a collateral asset.

Users need to be educated and be aware of the risks of the markets they enter

Response

> Acknowledged, see our risk analysis blog post.

Euler performed a risk analysis on manipulating UniswapV3 TWAP oracles. In addition, Euler reduced the number of collateral tokens they start out with and are considering to provide protocol-owned liquidity across the entire price range for these UniswapV3 token pairs to make price attacks economically infeasible.

3.2 Flashloans are broken if fee is activated (medium)

The `FlashLoan._loan` function pulls in an amount equal to `amount + FEE` but then requires that the balance of the contract is equal to `amount` only, without the `FEE`. For (non-fee-on-transfer) tokens this will always make the flashloan fail if `FEE` is non-zero.

Recommendation

We recommend updating the balance check to `require(IERC20(token).balanceOf(address(this))>= amount + FEE, 'e/flash-loan/pull-amount');`

Response

> Fixed. We decided to just remove the FEE parameter altogether since we have no intention to ever charge this fee.

3.3 Strict balance check breaks flashloans (medium)

The `FlashLoan._loan` function performs a strict balance check on the pulled-in token amount:

```
1 require(IERC20(token).balanceOf(address(this)) == amount, 'e/flash-loan
    /pull-amount');
```

An attacker can send a tiny amount of tokens to the flashloan contract and this balance check will always fail as `require(IERC20(token).balanceOf(address(this))= grieferAmount + amount + FEE > amount`. The result is that flashloans will revert.

Recommendation

We recommend updating the balance check to `require(IERC20(token).balanceOf(address(this))>= amount + FEE, 'e/flash-loan/pull-amount');`

Response

> Fixed. The griefing attack is prevented in the way you suggest, thanks!

3.4 Imprecise interest accrual (low)

While the contracts use compound interest on a per-second basis to accrue the debt, the same interest rate is compounded each second in `BaseLogic.initAssetCache` by computing the new interest accumulator as `pow(1.0 + interestRate, deltaT)* oldInterestAccumulator`. This uses the same `interestRate` for each time step. Note that the interest rate is determined by the utilization and accruing debt at each time step would lead to a (slightly) higher utilization as the new interest is part of `totalBorrows`.

The actual interest is therefore under-reported. While recomputing the interest rate after each time step for the compounding computation might cost too much gas and the change in utilization rate might be small, the contract should update the interest rate after every user action to ensure that the approximation error remains low even in high-activity markets.

As an example, while the `DToken.transferFrom` function updates the `interestAccumulator`, it does not update the `interestRate` afterwards.

Recommendation

Consider updating the `interestRate` each time a new interest accumulator is set. (It's only updated when doing an `increase/decreaseBalance` or `increase/decreaseBorrow` call.)

Response

> Acknowledged. This is a really good point. Interest accrual impacts the total borrows which impacts the interest rate, in sort of a continuous feedback loop. So, as pointed out, to track this accurately we'd need to re-target the interest rate every second or (more realistically) find some

> sort of closed form solution. This closed form solution could just be a very slight increase of the interest rate.

> One down-side of this design is that the amount of interest earned isn't totally independent on the frequency the market is interacted with. Fortunately, the difference in earned interest will generally be quite small. In comparison, Compound ironically doesn't compound its interest in between interactions (it uses simple interest). The actual difference between per-second compounding and Compound's erratic-compounding is very small, and the deviation in our case will be smaller still. Additionally, we plan to implement reactive interest rates in the future which will have a much larger impact on the interest accrued relative to interaction frequency.

> The DToken transfer not re-targeting the interest rate is sort of an oversight. We will think about this a bit more and perhaps fix that (although it would increase the gas required for DToken transfers).

3.5 Flashloan fee griefing attack for existing approvals (low)

If a contract approves the `FlashLoan` contract with more than their flashloan & fees, anyone can perform a griefing attack which will lead to the approver losing tokens equal to the fees.

This is because the flashloan `receiver` is not authenticated and anyone can start flashloans on behalf of another contract.

POC

1. Call `FlashLoan.flashLoan(receiver=victim, ...)`.
2. Loan amount + fees will be transferred from the `receiver` in `_loan`: `Utils.safeTransferFrom(token, address(receiver), address(this), amount + FEE);`

If fees are non-zero, it's possible to drain the victim's balance if their contract is implemented incorrectly without proper authentication checks.

Recommendation

This is an inherent issue with EIP-3156 which defines the interface with an arbitrary `receiver`. Contracts should be aware to:

1. revert if the flashloan was not initiated by them and
2. never approve more flashloan tokens to be returned than they need to pay back.

Response

> This is good advice for users of EIP-3156, but there is nothing for us to do here. As noted this is part of how the specification works. In addition, our flash loan adaptor's fee amount is always 0.

3.6 Flashloan does not work well with fee-on-transfer tokens (low)

Certain ERC20 tokens make modifications to their ERC20's `transfer` or `balanceOf` functions. One type of these tokens are deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`.

The `FlashLoan` contract would send out the flashloan amount (`amount`) but only receive back `(amount + FEE)* (1.0 - transferFee)`. (More tokens will be lost in the `dToken.repay` call). Thus slowly losing tokens on each flashloan.

Recommendation

As this attack is expensive for the attacker (they receive the after-fees amount but have to pay back the full amount) the impact is considered to be low. However, as arbitrary tokens can be listed on Euler, this is a possibility for malicious token owners with minting rights to screw over users by draining the pool.

One possible mitigation is to measure the asset change right before and after the asset-transferring calls (`Utils.safeTransferFrom(token, address(receiver), address(this), amount + FEE)` in `_loan`) to ensure that no tokens are lost.

Response

> Acknowledged. We will add a check into the flash loan adaptor to verify the expected amount was received.

3.7 Dangerous `FREEMEM` modifier (low)

Presently, the EVM memory grows linearly and allocating new memory space costs gas. The `FREEMEM` modifier is supposed to counteract this memory region growth by resetting the free memory pointer after the function has run to its initial values.

It's not obvious that this does not result in this already initialized memory region being reused and old data being read. While it usually should not happen that the EVM reads an uninitialized memory address, this modifier is an additional risk.

Recommendation

Consider not using this modifier to be able to reason about the calling functions better.

Response

> FREEMEM is measured to provide a small but non-zero gas savings. Although we consider it low risk, we will discuss whether to remove this for the production deploy, since none of the auditors seem to like it. One clarification for the audit description: This memory-recycling behaviour is entirely compatible with the EVM itself. The risk is that the solidity compiler or our own code may rely on newly allocated memory being all zeros (but that's what the 0xDEADBEEF development-mode check is supposed to uncover).

3.8 PToken/EToken/DToken does not emit Approval event in `transferFrom` (low)

The `PToken`/`EToken`/`DToken`'s `transferFrom` function does not emit a new `Approval` event when decreasing the allowance. Most ERC20 implementations, like OpenZeppelin's, emit this event when the `allowance` is decreased. Off-chain scripts and frontends will not correctly track the `allowance`s of users when listening to the `Approval` event. This can lead to failed transactions as a wrong, higher approval value is assumed.

Recommendation

Emit the `Approval` event also in `transferFrom` if the approval is decreased. (This has to be done in the token interface proxy, not the module, for `EToken` and `DToken`.)

Response

> This was an item on our TODO list. We did this for EToken and DToken a few days ago, and will do the same for PToken.

3.9 TWAP only works with ETH as a reference asset (low)

The `RiskManager`'s TWAP oracle is implemented with a generic `referenceAsset` even though `ETH` is used in practice. The `getPriceInternal` function is supposed to return the price in the reference asset with

an 18 decimal precision, see `computeLiquidityRaw`:

```
// @audit divides by 1e18
assetLiability = assetLiability * price / 1e18;
```

However, `decodeSqrtPriceX96` returns the price in `referenceAsset` precision in the first `if` branch:

```
price = FullMath.mulDiv(sqrtPriceX96, sqrtPriceX96, uint(2**(96*2)) / 1e18) / assetCache.underlyingDecimalsScaler;
// pricePrecision = (referenceAssetPrecision - underlyingPrecision) + 18 - (18 - underlyingPrecision) = referenceAssetPrecision
```

Recommendation

Add a comment that `referenceAsset` must be WETH or another 18 decimal asset or fix the code to work with assets with less than 18 decimals.

Response

> Acknowledged. The code currently assumes the reference asset is 18 decimal places. We will add this assumption to the documentation.

3.10 Special treatment of `type(uint256).max` value can lead to errors (minor)

In the `BaseLogic.withdrawAmounts` function, the special `type(uint256).max` value indicates to only withdraw the user's *current balance*.

This function is also used in swaps in `Swap.swapUniExactOutput` to make the user pay for the swap. If the required swap amount (`swap.amountIn`) for the desired trade-out amount returns this special value, only the user's current balance is reduced.

Recommendation

While it should not happen that a Uniswap trade returns a trade-in amount of the maximum possible value, it's better to check for any values with special meanings and revert if encountered in the swap.

Response

> Acknowledged. We will add a check to the swap module to detect this scenario.

3.11 Uniswap TWAP observation cardinality always increased (minor)

The `RiskManager.getNewMarketParameters` function always increases the UniswapV3's observation cardinality (price history slots), even if the desired `MIN_UNISWAP3_OBSERVATION_CARDINALITY` is already covered.

Recommendation

Check if the pool's next cardinality is already above `MIN_UNISWAP3_OBSERVATION_CARDINALITY`, only then increase it by the difference to reach the desired minimum cardinality. This saves gas.

Response

> This behaviour is by design. The *minimum* cardinality is enforced at market activation time. There is no maximum cardinality except what is specified by uniswap itself. Whenever the cardinality was insufficient to meet our configured TWAP interval, it is extended by 1.

3.12 Unused events (minor)

The `Transfer` and `Approval` events defined in `EToken` are never used as these events are emitted on the proxy and not in this module.

Recommendation

Remove unused code in `EToken` by removing the `Transfer` and `Approval` event definitions.

Response

> It's true that the events themselves are never emit'ed by the contract code. However, the generated ABI for EToken includes these events. In some of our integrations we are using the module ABIs to

> interface with the proxies, and removing those events might cause some minor breakage so we'd prefer to leave them in.

3.13 IRMLinearRecursive appears incomplete (undetermined)

The `IRMLinearRecursive` contract never sets the `prevUtilisation` and `prevInterestRate` storage fields but reads from them.

Recommendation

The desired behavior of this contract is unclear to the auditors. It seems to still be in development.

Response

> Acknowledged. This is an incomplete module that was used for testing. We will move this code to a separate branch until/if it's ready to deploy.

3.14 Miscellaneous (minor)

- `BaseLogic.isSubAccountOf`'s first argument is called `primary` which indicates that it is *not* a sub-account but a "real" address account. However, it works and is also used with subaccounts as the first parameter. Consider renaming the `primary` argument.

  > Acknowledged. The nature of the xor check means that there isn't really any difference between primary and sub-accounts in this function. The naming was mostly to encourage a convention which (as mentioned) is not totally followed anyway. We'll consider changing the parameter name.

- `Storage.AccountStorage` comment states `1 + 5 + 4 + 20 = 30` bytes as the structure's pack size but misses the 32 bytes of `averageLiquidity`

  > The comment is actually meant to reflect just the entries in the following packed slot, not in the entire struct. See the similar comment before the first slot in AssetStorage. We'll update the comment to be more explicit about that.

- Markets of `ETokens` (and `DTokens`) can be created which will create `ETokens` of `ETokens`. While no immediate issues were found there could be unintended side effects to two independent markets with their own market parameters tracking related underlyings. Reconsider if creating markets of existing `EToken`s is desired.

> I believe trying to create an “eeToken” or “edToken” will fail with “e/markets/invalid-token”. There is a test for this in test/activateMarket.js .

4 Conclusion

No critical issues have been found that would lead to an unexpected loss of funds for reasonable usage of the protocol. The nature of a *permissionless* protocol will always lead to issues, like price manipulation attacks, with certain tokens that might be controlled by malicious parties and could in the end lead to lenders and borrowers losing funds in liquidations or being left with bad debt. Therefore, it is advised to only lend and borrow tokens after having done due diligence on the tokens and ensuring that there's enough diversified liquidity in the UniswapV3 pools.

Overall, the documentation and the codebase are of high quality. The contracts are built using a comparably complex module-based approach with proxies acting as entry points. Low-level assembly instructions are used in these base components to save on gas costs and the general code complexity is quite high to accommodate the vast amount of protocol features.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.