FIAT DAO Delphi audit

Smart Contract Security Assessment

Mar 1, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of several smart contract modules of the FIAT DAO Delphi protocol.

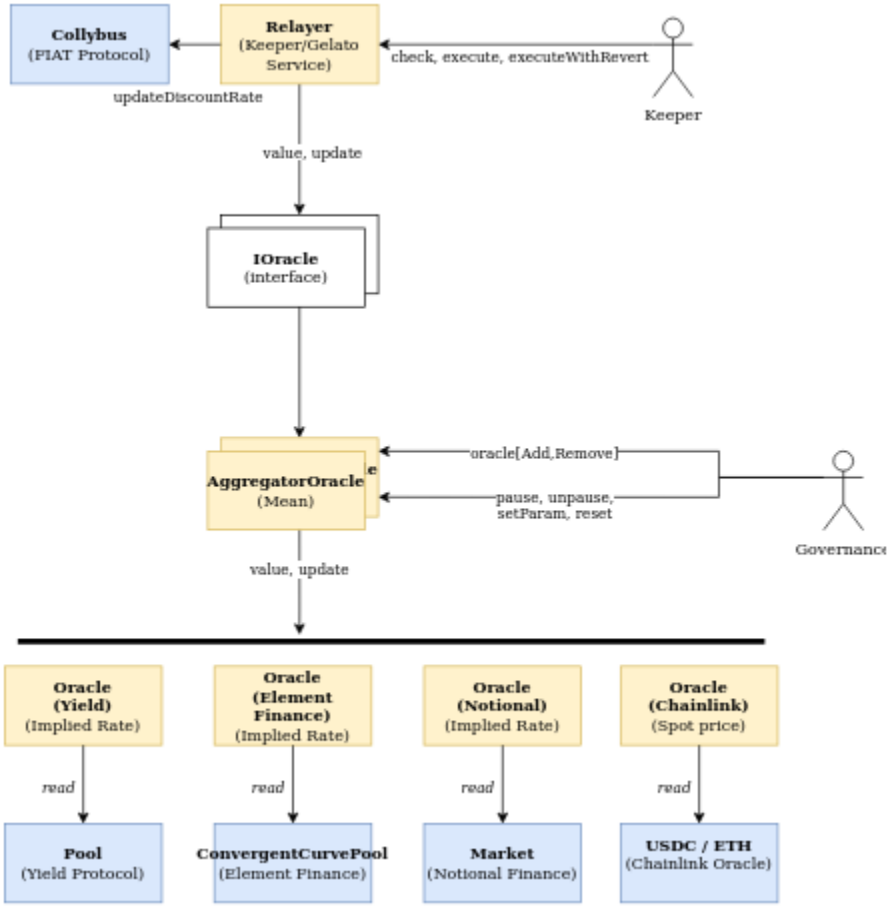
The scope of the audit included the most security critical parts of the at the time private repository https://github.com/fiatdao/delphi, up to commit a585f6c61ec616c6dfcebe756d21bcda1c6bb1d0. More specifically, the auditors examined in detail the contracts (excluding tests) under the following directories:

- aggregator
- guarded
- oracle
- oracle_implementations
- pausable
- relayer

The main auditing effort was 2 auditor-weeks. The auditors considered protocol-level attacks, the general pricing model, and other major considerations in the overall architecture. We reviewed the code in significant depth, assessed the economics of the protocol and processed it through automated tools. We also decompiled the code and analyzed it, using our static analysis (incl. symbolic execution) tools, to detect possible issues. On top of that, a domain expert studied and confirmed the security guarantees that the developers claim the protocol offers against price manipulation. An additional document containing the mathematical “proof” of this claim has been handed over to the protocol developers.

Setting and Caveats

The FIAT DAO Delphi protocol is an on-chain price feed for DeFi fixed interest rates. The core of Delphi is robust, with a clean architecture and reasonably simple, well-organized code. The protocol's architecture is described in the following diagram.



The core logic is cleanly split in the Oracle, AggregatorOracle and Relayer contracts, allowing for modularity but with limited code complexity. The abstract Oracle contract is implemented for different markets (e.g., Element Finance, Notional), which are sourced for market rates. The spot rates are fetched and then aggregated in Delphi oracles according to an exponentially-weighted moving average (EWMA). Market rates for the

same asset are aggregated via the AggregatorOracle contract, offering greater confidence and making price manipulation harder. The Relayer contract allows checking the oracles for rate/price changes and in case these are greater than a specified delta defined for each asset, they are pushed to Collybus, i.e., the main FIAT DAO protocol.

The Delphi protocol is continuously tested against a substantial number of tests. The audit did not consider the completeness of the test suite. Nevertheless, the auditors studied a number of tests to better understand the protocol and its parameters.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	The EWMA mechanism fails if updates are performed via Relay::executeWithRevert	**RESOLVED**

Oracle updates are performed in two layers: the `Oracle` performs the EWMA updates (`Oracle::update`), while the `Relayer` pushes the update when the Oracle's value is modified by at least a `delta` threshold (`Relayer::executeWithRevert`), i.e. when:

```
|newvalue - oldvalue| >= delta * oldvalue
```

If the change is smaller than the defined `delta`, the `Relayer::executeWithRevert` function reverts. According to the FIAT DAO team the reason is to perform the updates via the Gelato network only if there is a big enough change: Gelato will try this function locally on every block, and only really execute it on the blockchain when it does not revert. When the function reverts, the `Oracle::update` calls performed by this function will also revert, so `Oracle::update` will actually be executed only when the value changes by at least `delta/alpha%`! Not performing the updates, however, prevents the EWMA computation from working as expected.

As an example, let `alpha` = 0.2 (actual value used in test code) and `delta` = 0.05. Assume that the current actual value is 100 and is slowly increasing. After a long time, the actual value reaches 105 (`delta`=5% change). During this period, multiple EWMA updates could have happened such that the oracle value should have approached the actual one of 105. However, since *no update was really executed* due to `executeWithRevert` reverting, the `Oracle` value is still 100. Following the EWMA, In the next update the `Oracle` will update its value to

```
100 + (105-100) * 0.2 = 101
```

and the `Relayer` will think that the value changed by only 1%, less than delta=5%. No update will ever be executed until the actual value reaches 125 (25% change).

One might be tempted to simply use a smaller `delta`, e.g., 1% instead of 5%, to also account for `alpha`. But this only partially solves the problem. In the example above, when the actual value reaches 105 and the Oracle updates to 101, the Relayer will accept to perform the update (`delta` 1% is reached). However, the new value stored in the `Relayer` will be 101 and not 105 (the current actual value)! Even worse, assuming that the value stays stable at 105, in the next update the Oracle will update it to

101 + (105-101) * 0.2 = 101.8

moving closer to 105. But the Relayer will now see a change of only 101.8 - 101 = 0.79%, that is less than delta = 1%, so it will revert the update! No more `Oracle` updates will be ever performed (unless the current value increases even more), and the value will never converge to 105.

The problem is essentially that EWMA updates should be frequently performed, such that the `Oracle` value stays up-to-date. This should be independent from whether a `Relayer` update is needed or not. A solution could be to track in `Relay::executeWithRevert` whether any underlying Oracle update succeeded, and commit the transaction if it did (even if the `Relayer` does not update its own value).

Comment on the fixed version [this has now been addressed]:
`Relayer::execute` checks whether a collybus update is needed (`checkDeviation`) even if `oracleUpdated` is false. However, the `updated` flag is only set on oracle updates, not collybus updates!

It is not clear whether there can be a situation where a collybus update is possible without an oracle update (in the current code probably it is not).

- If it is possible, or it becomes possible in a future version (e.g., by allowing `minimumPercentageDeltaValue` to be modified), then not setting the `updated` flag is a problem, because the transaction will be reverted, although a collybus update was to be performed.
- If it is not possible, then it is preferable to avoid the collybus update check altogether when `oracleUpdated` is false, to make the code clearer and save gas.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Oracle::update possibly vulnerable to reentrancy	**RESOLVED**

`Oracle::update` has no reentrancy protection. As a consequence, if `Oracle::getValue` allows to call adversary-controlled code, an adversary could reenter multiple times into `Oracle::update`, and perform an arbitrary (limited only by gas) number of updates in the same block. This is because the timestamp check is performed before `Oracle::getValue`, but updated after the call (so the nested updates will be performed after all checks have succeeded).

```
function update() public override(IOracle) {
  if (lastTimestamp + timeUpdateWindow > block.timestamp) {
       return;
  }

  // we could reenter update() from here, after the timestamp check
  try this.getValue() returns (int256 returnedValue) {
      // oracle update happens here
       // nextValue = ...

      // timestamp is updated last
       lastTimestamp = block.timestamp;
  }
}
```

Performing multiple updates in the same block would completely nullify the EWMA protection against oracle manipulation.

We label the issue as “medium severity” simply because `Oracle::getValue` is unlikely to allow reentrancy in practice. Still, `Oracle::getValue` calls external functions, and supports a variety of different underlying oracles, so this possibility cannot be ruled out.

We recommend either to add a reentrancy guard, or to update the timestamp immediately after the check (and reset it if `Oracle::getValue` fails).

M2	Oracle updates can be performed by anyone	**RESOLVED**

`Relay::executeWithRevert` and `Oracle::update` are public functions callable by anyone. This is not necessarily a problem, but allowing updates only from authorized accounts has several advantages.

1. It essentially allows hiding the updates from the mempool. The project's documentation mentions the use of private transactions as a defense measure against Oracle manipulation:

   Security Measures

   Oracle update tx is relayed through Bloxroute so no bundling (bc update tx never enters public mempool)

   For this to work, however, the update functions need to be callable only by authorized accounts. Otherwise an adversary can perform their own updates as part of an attack, even though the protocol's own updates are hidden from the mempool.
2. Even if the authorized update transactions are visible in the mempool, performing a sandwich attack against the authorized update is costly, since it requires to bribe the miner and to possess the required funds to manipulate the Oracle. On the other hand, if an adversary can perform the update themselves, then such attacks become much easier since they can be done in a single transaction. There is no need to bribe a miner to reorder different transactions, and the adversary can use flash loans to acquire large amounts of funds.
3. Public update functionality means that the `Oracle::timeUpdateWindow` parameter needs to be carefully tuned for each market. It needs to be great enough to protect against an attacker manipulating the market price and calling Oracle::update multiple times in a short period of time to defeat the purpose of the EWMA price mechanism. On the other hand, `timeUpdateWindow` needs to be small enough to allow the oracle to follow relatively closely the market price, especially when using EWMA. Private updates do not come with the risks of choosing a value for `timeUpdateWindow` that would allow an attacker to circumvent the security of the EWMA mechanism without enough manipulation power.

If there are no other factors that suggest the need for public updates, we recommend to perform them only from authorized accounts.

LOW SEVERITY:

ID	Description	STATUS
L1	The use of median (instead of mean) would be preferable in `AggregatorOracle`	**DISMISSED**

For certain types of adversaries, aggregating Oracles via their *median* (instead of their mean) has preferable properties. Consider, for instance, an adversary who can manipulate a *single* Oracle, but in an unbounded way (they can set the Oracle's value to an arbitrary value). An unbounded change to a single element produces also an unbounded change to the mean! Thus no extra security is provided by the aggregator in such a scenario.

On the other hand, unbounded changes to a single value can only affect the median in a very limited way: the real median can only be changed to its adjacent values (in the order). So if n-1 Oracles are accurate, manipulating a single one is perfectly safe.

Of course there are other scenarios in which the mean performs better than the median. For example, by controlling n/2 oracles, the median is fully compromised, the remaining n/2 are not used at all in the computation, so the mean is a bit preferable in this case. However, we find that such scenarios are less realistic than the case of controlling a single Oracle; controlling n/2 Oracles is too powerful to defend against via aggregation alone anyway.

[The FIAT DAO team, being aware of the security tradeoffs of using the mean instead of the median value and of their system's requirements, has concluded that for the time being it is not necessary to change the current implementation.]

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	`Relayer::check` can be removed to save gas	**RESOLVED**
`Relayer:check` is called by `Relayer::executeWithRevert`, to revert the transaction if no updates are needed, duplicating most of the logic in `Relayer::execute`. It appears that the Gelato network [does not charge for transactions at the moment] (simulated or performed in the actual blockchain), thus it would be preferable to directly call `Relayer::execute`, which would return a flag showing whether an update was performed or not, and revert if the flag is false. • If the transaction reverts no charge is made, thus it is not important that the more expensive `execute` function was used instead of `check`. • If the transaction gets executed, the `execute` function will be called anyway, and the cost of calling `check` is saved.		
A2	The `OracleValue` event of `AggregatorOracle` can be more descriptive	**RESOLVED**
The AggregatorOracle contract defines and emits the `OracleValue(int256 value, bool valid)` event when a new value is fetched from an `Oracle`. Nevertheless, the oracle's address is not part of the event.		
A3	Floating version pragma in contracts	**INFO**
The floating version `pragma solidity ^0.8.0` is used allowing the contracts to be compiled with any version from 0.8.0 up to 0.8.12 of the Solidity compiler. Floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment, even though versions might not differ drastically.		

A4	Compiler bugs	INFO

The contracts can be compiled with the Solidity compilers v0.8.0-v0.8.12. At the time of writing some of these versions exhibit a subset of 4 known issues. We have reviewed the issues and do not believe them to affect the contracts. More specifically the known compiler bugs associated with the aforementioned Solidity compiler versions are:

- Memory layout corruption can happen when using abi.decode for the deserialization of two-dimensional arrays.
- For immutable variables of a signed integer type shorter than 256 bits, sign extension of its value is not always properly performed. According to the Solidity team the value can only be accessed in its unclean state when using inline assembly.
- The compiler does not correctly compute the storage layout of user defined value types for types that are shorter than 32 bytes, always using a full storage slot for these types, even if the underlying type is shorter.
- The bytecode optimizer will incorrectly re-use previously evaluated Keccak-256 hashes under certain scenarios.

[After our suggestions the FIAT DAO team has upgraded and fixed the compiler version to 0.8.12.]

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.