OpenZeppelin

Compound Open Price Feed – Uniswap Integration Audit

OPENZEPPELIN SECURITY | JULY 17, 2020

Security Audits

The Compound team engaged us to audit a new view for their Open Price Feed, featuring an integration with Uniswap V2. The audited commit is `d0a0d0301bff08457d9dfc5861080d3124d079cd`, and the following files were included in scope:

- `UniswapLib.sol`
- `UniswapConfig.sol`
- `UniswapAnchoredView.sol`

High-level overview of the system

The new view for the Open Price Feed intends to allow a configurable set of asset prices to be posted on-chain by a trusted reporter, which are anchored by asset prices fetched from Uniswap V2 markets. Posted prices can only deviate from Uniswap V2 prices to a certain extent (defined by the deployer of the view), greatly limiting the reporter's powers to manipulate the oracle. It must be noted that, as a difference from our previous audit of the Open Price Feed, in this case the reporter is represented by a single account, and the price is *not* calculated as a median. In other words, the oracle's official price for the set of assets is dictated by a single reporter account, though always bounded by the anchor.

The reporter account is set during construction, and cannot be later modified. Yet the view allows the reporter to invalidate itself, and from that moment on all prices will forever be updated using the on Uniswap's new time-weighted average prices. As a result, the reporter not only is limited to manipulate the Open Price Feed price due to the anchoring mechanism, but also should not be able to simply manipulate the anchor itself, thanks to integrity mechanisms inherent to Uniswap's time-weighted average prices introduced in the latest version of the protocol (further detailed in Uniswap's documentation and audit).

To understand how price anchors are calculated, a few points must be noted:

- The *minimum* period over which time-weighted average prices are calculated before being set as anchors in the Open Price Feed view is dictated by the immutable `anchorPeriod` state variable of the `UniswapAnchoredView` contract. This applies to all assets handled by the view.
- The implemented mechanism is based on a "lagging window" that will calculate average prices over varying periods of time, ranging between `anchorPeriod` and `2 * anchorPeriod` seconds. Therefore, the `anchorPeriod` value should be chosen sensibly, considering that longer periods will make it harder (and more expensive) for an attacker to manipulate a time-weighted average price in a Uniswap market (and therefore the anchor used by the view), but might result in less up-to-date prices (as described in Uniswap V2 whitepaper).
- It is possible that the average price used as anchor is calculated over a period of time *larger* than `2 * anchorPeriod` seconds if the reporter account takes more time than expected between subsequent price reports.
- The mechanism works as expected as long as sufficient time passes between the view's deployment and the first price report. This is acknowledged by Compound, and we describe additional details in **"[L02] Immature time-weighted average prices could be used as anchors"**.

One final point to highlight is that there are asset prices not expected to be posted by the reporter. Instead, they are to be fixed at the view's deployment. This is the case for USDC, USDT and SAI. While the price of the first two is expected to be fixed in US dollars, the price of SAI is going to be fixed in ETH. Its price in US dollars will therefore be subject to changes depending on the reported price of ETH in US dollars.

Critical severity

None.

High severity

None.

Medium severity

[M01] Convoluted price update mechanism once reporter is invalidated

The reporter account in charge of posting prices to the Open Price Feed's view can invalidate itself calling the `invalidateReporter` function.

Once the reporter is invalidated, prices are still to be posted to the view via the `postPrices` function so that they are updated with the asset's corresponding anchor value (regardless of whatever price the caller reports).

However, the price for an asset can only be updated with the anchor value if the messages and signatures sent along the call to `postPrices` belong to the reporter (which was invalidated). If the messages sent were not correctly signed by the invalidated reporter, the `postPrices` function will not update the asset's price. In other words, after the reporter is invalidated valid messages signed with the reporter's presumably compromised key are still needed to keep the view up-to-date with the latest prices.

Note that should the compromised key be no longer accessible, whoever calls `postPrices` will need to use old messages signed with the compromised key. If the key is lost before at least one such message is signed, it will be impossible to post prices to the view, which will need to be re-deployed.

Consider modifying the `postPrices` function so that signed messages are no longer required to update the view once the reporter account is invalidated.

[M02] Inconsistent behavior when anchor price is zero

V2 market for the asset.

Should the anchor price be zero, it will be impossible for the reporter to match the anchor price and effectively set the asset's price to zero. This is due to the fact the reporter's price must be greater than zero to be considered within the anchor – a check that is always done as long as the reporter is not invalidated.

When a reporter is invalidated (after executing the `invalidateReporter` function), the `UniswapAnchoredView` no longer takes into account prices submitted by the former reporter account, and considers the price from Uniswap as the "official" price for the asset. In this case, if the anchor price is zero and the reporter has been invalidated, the asset's price can effectively be set to zero without any restrictions.

To favor consistency, consider defining a single behavior for the oracle when the anchor prize is zero. In particular, the oracle should allow (or disallow) asset prices of zero regardless of whether the reporter has been invalidated or not.

[M03] Unexpected behavior when retrieving the underlying price of a cToken

The `getUnderlyingPrice` function of the `UniswapAnchoredView` contract intends to implement the interface defined in the `PriceOracle` contract of the Compound protocol. While the functions' signatures correctly match, the implemented function does not behave as specified in the interface's docstrings. In particular, the interface states that it should return zero when the price is unavailable. However, the `getUnderlyingPrice` function of the `UniswapAnchoredView` contract will revert when there is no price registered in the Open Price Feed for the queried cToken address.

While this does not pose a security risk for the Open Price Feed, it is a deviation from the interface's expected behavior, which can cause unexpected errors when interacting with the oracle. Moreover, it should be noted that this particular case is not being tested in the contract's test suite. Relevant unit tests are to be included to comprehensively specify the function's expected behavior.

being emitted in the `pokeWindowValues` function using incorrect values. In particular, as it is being emitted *before* relevant state changes are applied to the `oldObservation` and `newObservation` variables, the data logged by the event will be outdated.

Consider emitting the `UniswapWindowUpdate` event *after* changes are applied so that all logged data is up-to-date.

Low severity

[L01] Initial configured markets might not be Uniswap V2 pairs

The entire Open Price Feed view based in the `UniswapAnchoredView.sol`, `UniswapConfig.sol` and `UniswapLib.sol` files work under the assumption that all market addresses set during construction behave as Uniswap V2 pairs. However, this assumption is never programmatically enforced in the audited contracts. Markets set during construction should only comply with the `IUniswapV2Pair` interface, and their final addresses are entirely dictated by the deployer of the `UniswapAnchoredView` contract.

Once the view is deployed, users can easily verify that the markets configured are indeed Uniswap V2 pairs by querying the `getTokenConfig` function with indexes from 0 to 29, ensuring the received addresses are listed in uniswap.info.

If this is the system's expected behavior, consider explicitly stating it in contracts or external, user-friendly, documentation. Otherwise, consider programmatically enforcing that market addresses set during construction are indeed the expected Uniswap V2 pairs.

[L02] Immature time-weighted average prices could be used as anchors

The mechanism implemented to use time-weighted average prices from Uniswap markets as anchors works as expected under the assumption that sufficient time will pass (at least `anchorPeriod` seconds) between the contract's deployment and the first time a price is posted by the reporter. Yet it must be noted that this restriction is never programmatically enforced. The `UniswapAnchoredView` contract allows the reporter to post prices as soon as the contract is deployed, which could lead to using time-weighted average prices calculated over dangerously short periods of time.

highlighting it. Moreover, the scenario is not being tested in the contract's test suite.

To favor explicitness, consider adding the necessary logic to prevent prices from being posted before enough time passes since deployment. Alternatively, should this suggestion not be viable in terms of gas costs, consider at least explicitly documenting the assumption in the contract, and adding related unit tests.

[L03] Implicit structure for reporter invalidation message

The `invalidateReporter` function of the `UniswapAnchoredView` contract implicitly assumes that the passed message will have a specific structure so as to decode it. The function expects a message made up of a string and an address, though never making it explicit in the external specification provided during the audit nor in the function's docstrings.

Given that any attempt to decode a message not following the expected structure will immediately revert the call, consider explicitly documenting the exact structure the message must have to effectively invalidate the oracle's reporter.

[L04] Unnecessary event emission after replay of reporter invalidation

The sole reporter of the `UniswapAnchoredView` contract can be permanently invalidated calling the `invalidateReporter` function, as long as a valid message (signed by the reporter) is provided. Upon success, the function sets the `reporterInvalidated` flag to `true`, and emits a `ReporterInvalidated` event. However, it must be noted that anyone can replay a successful call to the `invalidateReporter` function, even if the `reporterInvalidated` flag is already `true`. In this scenario, successful calls will unnecessarily emit the `ReporterInvalidated` event every time.

Following the “fail early” principle, and to avoid unnecessary event emissions that may cause confusion in off-chain clients tracking them, consider reverting any call to the `invalidateReporter` function once the `reporterInvalidated` flag has been set to `true`.

[L05] Not failing early when posting a non-reportable price

not to be reported but rather remain fixed (either in ETH or US dollars) since construction. Assets whose prices are reported are tied to a Uniswap market address used to fetch anchor prices, while those that are not to be reported are not associated with any market.

The `postPrices` function does not explicitly validate whether a received price is expected to be reported or not. As a consequence, it takes several operations until the transaction is effectively reverted. This ultimately occurs down the call chain when attempting to fetch the anchor price for the asset from a non-existing Uniswap market at the zero address in line 45 of `UniswapLib.sol`.

For added readability and explicitness, consider validating that posted prices via the `postPrices` are indeed expected to be reported.

[L06] Potentially unsafe arithmetic operations

In lines 113, 125, and 217 of `UniswapAnchoredView.sol`, potentially unsafe arithmetic operations might be performed when dividing by `config.baseUnit`, as this variable is defined in each token configuration provided during deployment and can equal zero.

To avoid undesired behaviors, consider using the `div` function provided by the OpenZeppelin Contract's SafeMath library, or consider adding a check in the contract's constructor to ensure that the `baseUnit` field of passed `TokenConfig` structs cannot be zero.

[L07] Precision loss in fixed point library

The `FixedPoint` library implements two functions for handling fixed point binary numbers represented in UQ112x112 format. The `decode112with18` function intends to decode a formatted number into a regular uint256 type with 18 decimals in precision. Since simply scaling up the UQ112x112 number could result in an overflow, the function uses an approximation to safely scale the number. Therefore some precision is lost in the resulting number. As an example, the division of two equal numbers does not yield `1000000000000000000` but `1000000000000002239` instead.

Consider adding thorough unit tests for the `FixedPoint` library to ensure it behaves as expected. Moreover, consider documenting any precision loss in the library, even if it does not

[L08] Shortcomings in testing practices

While doing a general review of the test suite of the project, a number of shortcomings were identified. In particular:

- There is no automated test coverage report. Without this report it is impossible to know whether there are parts of the code never executed by the automated tests; so for every change, a full manual test suite has to be executed to make sure that nothing is broken or misbehaving.
- There are relevant contracts and functions of the system that lack of unit tests. Some examples of this are the `invalidateReporter` function, some sections of the `postPrices` function (specifically when calculating the anchor price of other assets aside from ETH), and the entire `FixedPoint` library.
- Both unit and integration tests are placed in the same `tests` directory and are executed together.
- After following the instructions in the README file of the project, tests may fail due to Docker configuration errors that are not contemplated in the instructions.
- There are modifications in the architecture of the code only implemented for testing purposes, such as setting the `fetchAnchorPrice` function as `virtual` in order to override its functionality in the `MockUniswapAnchoredView` mock contract.

Having a healthy, well documented and comprehensive test suite is of utter importance for the project's overall quality, helping specify the expected behavior of the system and identify bugs early in the development process.

Consider thoroughly reviewing the test suite to make sure all tests run successfully after following the instructions in the README file. Introducing an automated code coverage report is highly advised, so as to ensure all relevant functionality is rigorously tested, taking into consideration that code should only be merged if it neither breaks the existing tests nor decreases coverage. Additionally, consider running the unit tests and integration tests separately, defining different environments for each of them.

[L09] Lack of indexed parameters in events

[L10] Lack of event emissions

The `constructor` of the `UniswapAnchoredView` contract does not emit the `UniswapWindowUpdate` event after setting the timestamps and cumulative prices for each token configuration given in the `configs` array.

As the initialization of these variables is of utter importance, consider emitting the `UniswapWindowUpdate` event.

[L11] Missing docstrings

All functions in the `UniswapConfig` contract lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

[L12] Erroneous comments

Inline comments in lines 24 and 27 of `UniswapAnchoredView.sol` mention a "median price", but the contract never calculates the median price of assets. Consider updating these comments to avoid misleading users and developers.

[L13] Constructor's parameter order does not match specification

The order of the parameters of the `UniswapAnchoredView` contract's constructor does not match the provided specification. In particular, according to the specification, the parameter `anchorToleranceMantissa_` should come *after* the parameter `anchorPeriod_`.

Notes & Additional Information

[N01] Prone-to-error implementation in UniswapConfig contract

The `UniswapConfig` contract stores important configuration related to each of the 30 tokens the oracle's view can handle. Instead of tracking all information for the 30 tokens in standard, more appropriate, data structures such as mappings or arrays, the Compound team has decided to store all data in individual state variables. As a result, the `UniswapConfig` contract has 242 internal state variables.

While the current implementation may be more efficient in terms of gas costs, the team has heavily compromised on readability and ease of maintenance, which we consider fundamental factors for the project's long-term sustainability. This kind of error-prone implementations is discouraged and should only be applied after thorough testing, so as to ensure gas-savings benefits far outweigh the loss in code quality.

[N02] Implicit upcasting of uint112 variable

To favor explicitness and code readability, consider explicitly casting the `denominator` variable from `uint112` to `uint224` in line 17 of `UniswapLib.sol`.

[N03] Code repetition

The `source` function of the `UniswapAnchoredView` contract appears to have been copied from the `OpenOracleData` contract. Similarly, the `mul` function appears to have been copied from the `AnchoredView` contract.

To avoid code repetition, favoring reusability and maintenance, consider factoring out the mentioned functions into libraries to ensure they are only defined in one place, and then import them as needed in the contracts.

[N04] Copied code from Uniswap V2 Periphery repository

The `UniswapV2OracleLibrary` library inside the `UniswapLib.sol` file is copied straight from the Uniswap V2 Periphery repository. To benefit from patches and new features in future with the GNU General Public License v3.0 under which the `UniswapV2OracleLibrary` is released.

[N05] Missing units

To avoid errors in future changes to the code base, consider using an inline comment to clearly state in which units the `anchorPeriod` state variable is measured. Similarly, docstrings of the `getUnderlyingPrice` function should state the unit of the returned price.

[N06] Redundant boolean check

Line 160 of `UniswapAnchoreView.sol` explicitly compares a boolean value to `true`. This is a redundant operation because the result will be equivalent to the boolean value itself. Consider removing the redundant comparison.

[N07] Constants not declared explicitly

There are several occurrences of literal values with unexplained meaning:

- In `UniswapLib.sol`: lines 27 and 112.
- In `UniswapAnchoredView.sol`: lines 77, 78, 79, 125, 174, 197 and 219.

Literal values in the code base without an explained meaning make the code harder to read, understand and maintain, thus hindering the experience of developers, auditors and external contributors alike.

Developers should define a constant variable for every magic value used (including booleans), giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them.

[N08] Naming

In the `UniswapAnchoredView` contract:

- The `acc` field of the `Observation` struct should be renamed to make it more self-explanatory.
- The `UniswapWindowUpdate` event should be renamed to `UniswapWindowUpdated`.
- The `AnchorPriceUpdate` event should be renamed to `AnchorPriceUpdated`.
- The `decoded_message` variable should be renamed to `decodedMessage`.

In the `UniswapConfig` contract:

- The `i` parameter in the `getToken` and `get` functions should be renamed to `index`.
- The `get` function should be renamed with a more self-explanatory name.

[N09] Inconsistent use of `uint` and `uint256` types

Throughout the `UniswapConfig` contract, the `uint` and `uint256` types are used interchangeably to declare 256-bit unsigned integers. To favor consistency, consider following the same convention on all these declarations.

[N10] Typographical errors

In line 193 of `UniswapAnchoredView.sol`, “unsiwap” should say “Uniswap”.

[N11] Lack of explicit visibility in state variables

The constant state variables `ethHash` and `rotateHash` of the `UniswapAnchoredView` contract are implicitly using the default visibility.

To favor readability, consider explicitly declaring the visibility of all state variables.

[N12] Unnecessary public visibility in functions

The `price` and `getUnderlyingPrice` functions of the `UniswapAnchoredViewContract` are defined as `public` but are not being accessed from within this contract. Consider changing their visibility to `external`.

Conclusions

quality.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin

About us
Jobs
Blog

© Zeppelin Group Limited 2023
Privacy | Terms of Use