PUBLIC

Code Assessment

of the Aggor Smart Contracts

July 26, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Chronicle with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Aggor according to Scope to support you in forming an opinion on their security risks.

Chronicle implements Aggor, an oracle aggregator: Values from a Chronicle oracle and another provider (Chainlink or UniswapV3) are combined into a single value and stored as a tuple (value, age). Several interfaces for authenticated read of this tuple are exposed.

The most critical subjects covered in our audit are functional correctness, resilience against manipulation and effective on chain operation.

There is the possibility of a false sense of security being created due to the aggregation of different sources. Issues uncovered discuss how the aggregated price might be manipulated. Specifically, the design decision to fall back to the value from the oracle that is closer to the previously stored value, when both sources deviate by more than the permissible spread, may lead to unexpected implications.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	6
• Code Corrected	4
• Risk Accepted	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Aggor repository based on the documentation files.

The scope consists of the following solidity smart contracts:

1. src/Aggor.sol

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	6 July 2023	e4f1f666b3758b0df5f0d3479e0a53c069ab3ee8	Initial Version
2	24 July 2023	d2534c1e392c855de2f4114468bdf147c631e2c1	After Intermediate Report

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies (e.g. UniswapV3 is assumed to work correctly as documented), and configuration files are not part of the audit scope. Notably the OracleLibrary imported from Uniswap is not part of this review and is assumed to be correct.

3 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Aggor is an oracle aggregator: Values from a Chronicle oracle and another provider (Chainlink or UniswapV3) are combined into a single value and stored as a tuple (value, age). Several interfaces for authenticated read of this tuple are exposed. The update of the (value, age) tuple can be triggered without requiring permission.

Wards are roles governing Aggor (initialized with `msg.sender`) with the privilege to call:

- `rely()` - adds a new address to wards.
- `deny()` - removes an existing address from `wards`.
- `kiss()` - gives reading access to an address by adding it to `_buds`.

- `diss()` - revokes reading access of an address by removing it from `_buds`.
- `setStalenessThreshold()` - updates the staleness threshold parameter.
- `setSpread()` - updates the spread parameter. It's used to identify a possibly breached oracle if the discrepancy between oracle values exceeds this specified amount.
- `setUniswap()` - Update the Uniswap oracle. Disable and switch to Chainlink by setting to `0x0`.
- `setUniSecondsAgo()` - Sets the Uniswap TWAP period.

`poke()` (and the optimized variant) are the permissionless entry points of the contract for anyone to trigger an update of the value. This works as follows:

1. Reading from Chronicle: Queries the value. Checks whether the value is valid and not stale. Execution reverts if an invalid value is detected.
2. Reading from the second source: If no UniswapV3 pool is set the value is read from Chainlink, otherwise the Uniswap pool is the second source.
    - Reading from Chainlink: Query `latestRoundData()`, determination whether value is valid (non-negative, non-zero, not stale) and conversion to the decimals required for Aggor.
    - Reading from UniswapV3: Done using the OracleLibrary provided by Uniswap. Value is checked to be non-zero and converted to the decimals required for Aggor.

    It is ensured that reading the oracles was successful and resulted in a valid value. Otherwise the execution reverts.
3. The difference between the oracles is computed.
    - If the difference is within the accepted spread, the value is updated with the mean of both queried oracle values.
    - If the difference exceeds the spread, the value is updated with the value closer to the last value stored.
4. The age is set to the current `block.timestamp`.

The following view functions are provided for authenticated reading of the value and age:

- `read()` - returns `_pokeData.val`, and reverts if it is zero.
- `tryRead()` - returns `_pokeData.val` and a boolean flag indicating if it is zero.
- `read()` - returns `_pokeData.val` and `_pokeData.age`, and reverts if val is zero.
- `tryReadWithAge()` - returns `_pokeData.val` and `_pokeData.age`, together with a flag indicating if val is zero.
- `latestRoundData()` - reading entrypoint for Chainlink compatibility, where `roundId` and `answeredInRound` are always `1`. `startedAt` is always `0`. `answer` and `updatedAt` are `int(uint(_pokeData.val))` and `_pokeData.age` respectively.

The following view functions are also provided to retrieve the information about the configuration, wards and buds:

- `authed(address)` - returns if the address is a ward.
- `authed()` - returns an array of existing wards. Note that there could be duplicates address in the returned array if one ward is removed and added back later.
- `wards(address)` - returns an uint indicating if the address is a ward.
- `tolled(address)` - returns wether the address is in buds.
- `tolled()` - returns an array of existing buds, which could include duplicates as `authed()`.

- `bud(address)` - returns an uint indicating if the address is in buds.
- `chronicle()` - returns the address of the chronicle pricefeed.
- `chainlink()` - returns the address of the chainlink pricefeed.
- `uniPool()` - returns the address of the Uniswap pool set - or `0x0` if no pool is set.
- `uniBasePair()` - returns token0 of the Uniswap pool set - or `0x0` if no pool is set.
- `uniQuotePair()` - returns token1 of the Uniswap pool set - or `0x0` if no pool is set.
- `uniBaseDec/uniQuoteDec()`- returns the decimals of the token - or 0 if no pool is set.
- `uniSecondsAgo()`- TWAP window when querying the uniswap oracle
- `minUniSecondsAgo()`- minimum TWAP window set in the constructor
- `decimals()` returns the decimals of the pricefeed. This is currently a constant set to 18.

3.1 Trust Model & Roles

Wards are the privileged roles governing the contracts. They are fully trusted to set all parameters and feeds honestly and correctly. A malicious ward can seriously harm the security of the contract in the following ways:

1. wards can set arbitrary addresses as the UniswapV3 pool. If the address is a malicious contract, it could return an arbitrary price feed.
2. wards can set the `spread` limit to 100%, invalidating users assumption of outlier resistance.
3. wards can set the staleness threshold to a short value that might cause querying of the oracle to revert because of stale values, if `poke()` was not called in that same block
4. wards can arbitrarily `diss()` buds, causing their access to the oracle to fail

If Chronicle choses to implements wards as a timelock contract, the risk of these actions on users of the contract is greatly mitigated.

Buds: Role with permission to read from the pricefeed. No privileges for state changing functions on the contract.

Underlying pricefeeds (Chronicle, Chainlink, UniswapV3): Generally untrusted. Notably there is an important assumption: If the value between both sources differs by more than `spread`, the source reporting a value closer to the old value stored is used for the update.

Chronicle pricefeeds must return the value in 18 decimal representation.

Prices (in 18 decimal representation) must be guaranteed to be `<= type(uint128).max`.

No permission is required to trigger the update of aggregated value.

3.2 Changes in Version 2

- The Uniswap Oracle must now be set upon deployment and can no longer be updated. If an Uniswap pool has been configured, the priviledged role can switch the secondary source between Chainlink and Uniswap using the new `useUniswap` function.

4 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

6 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

6.1 Malicious Oracle Feed Can Hijack Aggor Price Aggregation

Security Low Version 1 Risk Accepted

CS-CAG-003

If one of the two selected oracles is malicious, it can take whole control of the Aggor price feed, despite the `diff > spread` check.

The behavior of Aggor is that, in case of disagreement between oracles, it uses the price that has changed the least since last update as the true one. A malicious oracle can in the span of two blocks (could be more) take control of Aggor by exploiting this behavior as follow:

1. move the malicious oracle price to the limit of `diff > spread`, so that the condition is not triggered but a small fluctuation will be enough to trigger it. The Aggor value will then be the mean of the two oracles.
2. in next block, if the true price feed decreases the spread, then the malicious oracle moves in the opposite direction to keep the spread on the limit of `diff > spread`, repeating step 2. If the true price feed increases the spread, the malicious oracle doesn't update its value: its value will now be the closest to the old one, and on next updates it can move the value however it wants as long as the distance doesn't increase more than the current distance with the true value.

This means that a malicious oracle can arbitrarily hijack the value of the Aggor feed, as long as the price growth/fall is not faster than exponential. In n blocks the price difference between the true value and the hijacked value can be up to `± (initial_diff)*2^n`.

In the practical context of Uniswap manipulation, which is cheap to execute on low liquidity pools, where a malicious actor would need to control at least 2 (possibly more) consecutive blocks in order to manipulate the price in Aggor to arbitrary values.

Risk accepted:

Chronicle acknowledges the risk, and states that the following mitigating factors are sufficient:

1. source oracles are inherently hard to manipulate, especially for a long time
2. the `spread` parameter guards against large manipulations in a single update

6.2 Oracle That Stops Updating Will Control Aggor Price Feed

Design Low Version 1 Risk Accepted

CS-CAG-004

If one of the two oracles stops updating, the value reported by Aggor will be the stuck value. This is because when the percentage difference between the two oracles become more than `spread`, the new value is chosen as the one of the two oracles which is closer to old value, which used to be the average of the two feeds. Since the distance has increased, but the stuck oracle hasn't changed value, the stuck value is closer to the previous mean than the other value. This means that if an oracle is "stuck" and the difference with the other feed grows to more than `spread`, the Aggor oracle will always return the stuck value. Stuck updates have been an issue in the past with oracles, for example when Chainlink paused the update of LUNA .

Risk accepted:

Chronicle is aware of this behavior. The risk associated is deemed similar to issue 6.1, the same mitigations and assumptions apply.

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Uniswap Oracle Order of Tokens Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	4

- Emitting Events and Reverting Code Corrected
- Ineffective diff != 0 Code Corrected
- Uniswap Price Can Overflow Maximum Value in 128 Bits Uint Code Corrected
- UniswapV3 Oracle Might Need Initialization Code Corrected

7.1 Uniswap Oracle Order of Tokens

Design High Version 1 Code Corrected

CS-CAG-001

Function `setUniswap()` does not allow to specify the order of the tokens in the price computation. The quoted price will be an amount of `token1` for a unit of `token0`. This ordering does not necessarily matches the order of the tokens in Chainlink or Chronicle oracles. In Uniswap v3 pools the ordering of `token0` and `token1` is dictated by the relative ordering of their addresses, while in Chainlink the price is always quoted in terms of USD or ETH.

Since there is no way to specify an alternative order of tokens for the Uniswap price calculation, the contract will be invalid for many token pairs.

Code corrected:

The uniswap pool is now set in the constructor, and a boolean argument called `uniUseToken0AsBase` allows to select one of `token0` or `token1` as the base token.

7.2 Emitting Events and Reverting

Design Low Version 1 Code Corrected

CS-CAG-002

The internal functions reading from external pricefeeds (`_tryReadUniswap()`, `_tryReadChronicle()`, `_tryReadChainlink()`) upon detecting an error emit an event and return `return (false, 0);` to the calling function. This function then reverts.

Note that events emitted in transactions (calls) that revert are not observable. Hence such events cannot be filtered for nor be observed in the common block explorers. The intended use of these events is unclear.

Code corrected:

Events preceding reverts have been removed.

7.3 Ineffective `diff != 0`

Design | Low | Version 1 | Code Corrected

CS-CAG-006

In `_poke()`, when determining if the oracle values differ by more than the acceptable spread, the condition `diff != 0` does not have any effect on the overall branch condition.

```
if (diff != 0 && diff > spread) {
        ...
  } else {
        ...
  }
```

If `diff` is 0, the condition `diff > spread` with both variables being unsigned integers, can never be true, hence the additional condition `diff !=0` is redundant.

Code corrected:

`diff != 0` has been removed.

7.4 Uniswap Price Can Overflow Maximum Value in 128 Bits Uint

Security | Low | Version 1 | Code Corrected

CS-CAG-005

The codebase assumes that the value returned by `_tryReadUniswap()` is smaller than `type(uint128).max`. This is not the case, as the maximum UniswapV3 token ratio (`token1/token0`) is `2**128`. This translates to a price, with 18 decimals, of `2**128 * 10**18`, which exceeds the maximum `uint128` value.

It is unlikely that the price for a valid pool and currency pair would ever be so high, but in casse a low liquidity pool is configured as the UniswapV3 source in Aggor, an attacker could manipulate the price upward. `_tryReadUniswap()` may validate that the value read is within safe bounds.

Code corrected:

A check has been added to ensure the price cannot overflow.

7.5 UniswapV3 Oracle Might Need Initialization

Correctness | Low | Version 1 | Code Corrected

CS-CAG-001

UniswapV3 pools expose the `observe(uint32[] secondsAgo)` external function to access the cumulative price and liquidity at given times before the current block timestamp. However, the `observe` function will revert if the earliest observation stored in the pool is before than the requested time. A pool can store up to `65535` observations, corresponding to approximately 9 days at 12 seconds per block, but the observation slots need to be made available through the pool's `increaseObservationCardinalityNext(uint16)` method. By default, pools are initialized with a single observation slot available. Care must be taken by the deployer in assuring that the selected Uniswap pools have a sufficient amount of observation slots available to record values as old as Aggor's `uniSecondsAgo`.

Code corrected:

`setUniswapSecondsAgo()` now ensures that `observe()` can be called successfully with the new value for `secondsAgo` by attempting to read from the Uniswap pricefeed.

7.6 N > Dec Condition in LibCalc.scale()

Informational | Version 1 | Code Corrected

CS-CAG-002

An ineffective require statement is present in LibCalc `scale()`:

```
function scale(uint n, uint dec, uint destDec)
    internal
    pure
    returns (uint)
{
    ...

    require(n > dec && n > destDec);

    ...
}
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Age Is Timestamp of Poke

Note Version 1

It's important to note that `_age` represent `block.timestamp` of the last poke. The age of the underlying values is older, up to the limit defined by the staleness period.

As poke is permissionless, the `_age` can be refreshed or advanced without any change in the actual value if there has been no changes in the source oracles.

8.2 Aggor Price Jumps When Spread Limit Is Crossed

Note Version 1

The spread limit guards against single outliers in one of the price feeds, making the aggregate price generally less volatile than the sources. However, if the values of the two feeds diverge slowly, the price will experience a sudden jump of approximately `spread/2` when the `diff > spread` is satisfied.

8.3 Immutable Feeds

Note Version 2

All underlying feeds are set as immutable in the constructor and hence these adresses cannot be updated. Should the need arise to change the addresses of these feeds a new instance of Aggor has to be deployed. Integrators of the pricefeeds must be able to switch to new implementation address.

8.4 UniswapV3 Price Manipulation

Note Version 1

If a low liquidity pools is used in Aggor, the cost of price manipulation of the internal uniswap oracle is drastically reduced. However, because of the technicalities described in Malicious oracle feed can hijack Aggor price aggregation an attacker will at least need to control 2 consecutive blocks (potentially more) to make their manipulation succesful, which mitigates the risk.