PRIVATE

Code Assessment

of the Aggor Smart Contract

August 08, 2024

Produced for

Chronicle

by

CHAINSECURITY

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. This updated report covers the latest changes.

Aggor, tailored for SparkLend, uses Chainlink and Chronicle as primary oracles. It features a tie-breaking mechanism that utilizes the Uniswap TWAP, also serving as a fallback option. Current oracle data is evaluated independent of historical values. Interfaces for authenticated access to read oracle data are provided.

The most critical subjects covered in our audit are functional correctness, resilience against manipulation and oracle data utilization.

For Aggor to operate correctly, it's essential that all parameters are correctly initialized upon deployment.

The general subjects covered are adherence to specification, usability and gas optimizations. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high 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	2
• Code Corrected	2
Medium-Severity Findings	0
Low-Severity Findings	5
• Code Corrected	3
• Acknowledged	1
• No Response	1

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:

```
src/Aggor.sol
src/IAggor.sol
src/libs/LibUniswapOracles.sol
```

In Version 4 the following file has been added:

```
src/libs/LibMedian.sol
```

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

V	Date	Commit Hash	Note
1	11 December 2023	f66a40320978e6a68360d6e2377440fa84c69885	Initial Version
2	08 January 2024	5a4c3fd66b1f62805ab489017e0e1af8e6896d21	After Intermediate Report
3	09 January 2024	18f0245f92e85d424c86bb1d804a03b64ff3ea36	Refactor Checks
4	05 August 2024	b80a638aff2f7fbc5cfb1ce84e823b8d0cf013b5	Updated Version

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.

The configuration of a deployed instance of `Aggor` is out of the scope of this review.

2.2 System Overview

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

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. This updated version is specifically implemented for the requirements of SparkLend. The chosen configuration is hardcoded and has two primary source oracles Chainlink and Chronicle with Uniswap TWAP as fallback oracle / tie breaker only. Upon being queried, the answer is determined based on the oracles, and contrary to previous versions of Aggor, there is no dependency on the state (previous values). Several interfaces for authenticated reads of the aggregator returning different levels of information (value, age, status information) are exposed.

Wards are roles governing Aggor (initial ward is set using the constructor parameter `initialAuthed`) 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`.
- `setAgeThreshold()` - updates the staleness threshold parameter. The new age threshold must be `> 0`.
- `setAgreementDistance()` - updates the spread parameter. It's used to identify a possibly breached oracle if the discrepancy between oracle values exceeds this specified amount. The new agreement distance must be `0% < agreementDistance<= 100%`.

Upon the oracle being queried, the determination of the price works as follows:

1. Reading from Chronicle and Chainlink. Checks on the returned value / status to determine whether the value is valid.
2. If both main Oracles returned valid values, check whether their answer is within the agreement distance.
    - If they are, the average is used as value. (Path 2)
    - If the oracles report values exceeding the allowed spread, the tie breaker is initiated. (If successful, Path 3)
3. If only one of the main Oracles (Chainlink or Chronicle) returns a valid value, this value is used. (Path 4)
4. If neither Chainlink nor Chronicle returned a valid value, if an Uniswap pool is configured, the Uniswap TWAP price is used. (Path 5)
5. If no value derivation is possible, a value and age of 0 is returned. (Path 6)

The tie breaker works as follows:

For pegged assets: The pegged asset price is added to the set of values [Pchainlink, Pchronicle, Ppeggedprice] and the median value is returned.

For other assets, an Uniswap pool must be configured. The Uniswap TWAP price is added to the set of values [Pchainlink, Pchronicle, Puniswaptwap] and the median value is returned.

The price determination results in a status of the following form:

```
Status({
    path
    goodOracleCtr
    badOracleCtr
```

```
    tieBreakerUsed
})
```

The following view functions are provided for authenticated reading of the oracle:

- `readWithStatus()` - returns the full status information
- `latestAnswer()` - returns the value (`int(uint(val))`).
- `latestRoundData()` - reading entrypoint for Chainlink compatibility, where `roundId` and `answeredInRound` are always 1. `startedAt` is always 0. `answer` and `updatedAt` are `int(uint(val))` and `block.timestamp` respectively.

Furthermore the following view functions are provided to retrieve 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 duplicate addresses 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 whether 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.
- `decimals()` returns the decimals of the pricefeed. This is currently set to the Chainlink decimals which is hardcoded to 8.
- `isPeggedAsset()` - returns `true` / `false` depending if the Aggregator is configured for a pegged asset or not.
- `peggedPrice()` - returns the set pegged price. If the Aggregator isn't for a pegged asset this value may be arbitrary.
- `uniswapPool()` - returns the Uniswap pool.
- `uniswapBaseToken()` - returns the base token of the Uniswap pool.
- `uniswapQuoteToken()` - returns the quote token for the Uniswap pool.
- `uniswapBaseTokenDecimals()` - returns the decimals of the Uniswap base token.
- `uniswapLookback()` - returns the lookback period when querying the TWAP oracle. Important parameter.
- `agreementDistance()` - returns the allowed spread between Chronicle and Chainlink. This value can be updated.
- `ageThreshold()` - returns the allowed age of the underlying oracle answer. This value can be updated.

2.2.1 Changes in Version 4

- Oracle Status no longer includes `badOracleCtr` and `tieBreakerUsed`.
- Pegged asset mode has been removed.

- The agreement distance is now set differently: Previously, the agreement distance was set as the maximum allowed deviation, e.g., a 5% deviation. Now, this distance is set based on the required level of agreement, e.g., 95% (100-5%).
- Changes in access control of read functions (Toll): An immutable address and 0x0 is granted read access, to ensure compatibility the full IToll interface is still implemented.

2.3 Trust Model & Roles

Oracle configuration (Chainlink, Chronicle, Uniswap) and important configuration parameters `isPeggedAsset` and `peggedPrice` are set during deployment and cannot be changed thereafter. Prior to Version 4 the different parameters for the Uniswap configuration are not enforced to be consistent, there is an assumption that the deployer sets them correctly. After deployment, before usage of the oracles the configuration should be carefully reviewed.

Note that some of these oracles (Chainlink, Chronicle) may be deployed behind a proxy and hence might change their behavior.

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

1. wards can update the `AgreementDistance`, invalidating users assumption of outlier resistance.
2. wards can set the age threshold to a short value that might cause querying of the oracle to revert because of stale values.
3. (Prior to Version 4) wards can arbitrarily `diss()` buds, causing their access to the oracle to fail.

If Chronicle chooses to implement 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 only one oracle is unavailable / reports an invalid value, the value reported by the other oracle is returned. Another important assumption is that `Aggor` is a `bud` in the `Chronicle` oracle so that it can read the price, otherwise `Aggor` will revert. Two subsequent separate calls to the TWAP oracle within Aggor are expected to result in the same outcome.

Chronicle pricefeeds must return the value in 18 decimals representation.

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

3 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.

4 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.

5 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:

- 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

5.1 Implementation Does Not Follow the Specification

Design Low Version 4

CS-AGGV2-006

Function `Aggor._read` does not follow the specification and queries the Uniswap TWAP twice when used as tie-breaker but the TWAP fails to return a valid value.

According to the specification the function must query the TWAP in case of the two oracles not agreeing on a value. If the TWAP is not available the function must return 0.

```
if in_agreement_distance(values):
    return median(values)  # Path: 2
else:
    twap_value := read(twap)
    if is_valid(twap_value):
        return median(values ++ twap_value)  # Path: 3
    else:
        # Error: No value derivation possible
        return 0  # Path 6
```

However, the implementation in `Aggor._read` does not return 0 when the TWAP is not valid, but instead continues the execution. Then it queries the TWAP a second time and finally returns 0 when the second call fails.

```
// If no oracle ok use TWAP.
(bool ok, uint128 twap) = _readTwap();
if (ok) {
```

```
    return (twap, age, Status({path: 5, goodOracleCtr: 0}));
}

// Otherwise no value derivation possible.
return (0, 0, Status({path: 6, goodOracleCtr: 0}));
```

Since it's expected that the call to the TWAP oracle results in the same outcome in both calls, the result returned by `_read()` is according to the specification.

5.2 UniswapV3 Oracle Might Need Initialization

Correctness **Low** **Version 1** **Acknowledged**

CS-AGGV2-004

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`.

Acknowledged:

Chronicle checks this in the deployment scripts:

```
This is checked in the deployment script via the UniswapV3 oracle library.
Specifically, it is checked that the twap/pool's oldest observation is older
than the `uniswapLookback` argument. Therefore, it is guaranteed that the
twap/pool supports the requested lookback.
```

In **Version 4** a check has been added in the smart contract:

```
// Verify TWAP is initialized.
// Specifically, verify that the TWAP's oldest observation is older
// then the uniswapLookback argument.
uint32 oldestObservation = uniswapPool_.getOldestObservationSecondsAgo();
require(
    oldestObservation > uniswapLookback_, "Uniswap lookback too high"
);
```

This check only ensures that the oldest observation at this point in time (during deployment of the contract) is sufficient.

To determine whether the TWAP lookup is always possible, it's important to consider the observation cardinality of the Uniswap V3 pool. Each pool has a fixed number of observation slots, determined by its observation cardinality, which limits the maximum number of observations that can be stored. Observations are recorded at maximum once per block. In low-activity periods, observations might be stored for longer durations, but during high activity, older observations are overwritten as new ones are recorded in each block. Therefore, it is important to ensure that the observation cardinality is sufficient to

support the desired `uniswapLookback period`, assuming a new observation is stored every block. This ensures that the required historical data will be available even in the scenario of maximum activity and hence, the TWAP oracle returning a price.

6 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	2

- Hardcoded Chainlink Decimals Code Corrected Acknowledged
- TWAP Price Decimal Mismatch Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	3

- Configuration Consistency Acknowledged Code Corrected
- Deprecated answeredInRound, Redundant Check Code Corrected
- Incorrect Check in SetAgreementDistance Code Corrected

Informational Findings	2

- Gas Optimizations Code Corrected
- Meaning of tieBreakerUsed Flag Code Corrected

6.1 Hardcoded Chainlink Decimals

Correctness High Version 1 Code Corrected Acknowledged

CS-AGGV2-001

The constant `_DECIMALS_CHAINLINK` is hardcoded to 8 regardless of the decimals of the `chainlink` oracle. This creates a decimals mismatch when the source is an `ETH` pair.

For example, the Chainlink price oracle for `stETH/ETH` has 18 decimals (https://etherscan.io/address/0x86392dc19c0b719886221c78ab11eb8cf5c52812#readContract#F3).

If `Aggor` is used for a `ETH` pair, `val_chr` will have 8 decimals and `val_chl` will have 18 decimals, this difference will most likely always trigger the tie breaker.

Code corrected:

Constant `_DECIMALS_CHAINLINK` has been replaced by an immutable `decimals`. This parameter is initialized during contract deployment based on data retrieved directly from the Chainlink oracle.

Acknowledged:

In Version 4, the decimals of the Chainlink pricefeed are no longer handled, it is assumed that the price returned is in 8 decimals, the same as Aggors decimals. Chronicle states that Aggor is only to be used for ETH/USD. If they ever were to use a quote token with Chainlink returning a price in different decimals, this will be adjusted.

6.2 TWAP Price Decimal Mismatch

Correctness High Version 1 Code Corrected

CS-AGGV2-007

This issue was reported by Chronicle.

The price returned by the TWAP oracle is in decimals of the quote asset which may not match Aggors decimals. If the decimals don't match, the TWAP price used as tie-breaker / fallback oracle may be off by an order of magnitude.

Code corrected:

The decimals of the TWAP price are now adjusted as necessary:

```
// Read twap.
uint twap = uniswapPool.readOracle(
    uniswapBaseToken,
    uniswapQuoteToken,
    uniswapBaseTokenDecimals,
    uniswapLookback
);

if (uniswapQuoteTokenDecimals <= decimals) {
    // Scale up
    twap *= 10 ** (decimals - uniswapQuoteTokenDecimals);
} else {
    // Scale down
    twap /= 10 ** (uniswapQuoteTokenDecimals - decimals);
}
```

6.3 Configuration Consistency

Design Low Version 1 Acknowledged Code Corrected

CS-AGGV2-002

1. Nothing prevents a zero `peggedPrice` if `isPeggedAsset` is true.
2. Nothing enforces the `uniswapBaseToken` and `uniswapQuoteToken` to actually match `token0` and `token1` of `uniswapPool_`.
3. The `uniswapBaseTokenDecimals` is not explicitly linked to the decimals of `uniswapBaseToken`.

Even though we expect the DAO to carefully review all the deployment parameters, we want to make Chronicle aware of these points.

Acknowledged:

Chronicle states:

```
We waive the checks onchain to reduce deployment costs. However, the checks
are incorporated into the deployment script.
```

Code corrected:

In Version 4, `peggedPrice` and `isPeggedAsset` have been removed from the codebase. Checks 2. and 3. are now enforced within the constructor of the `Aggor` contract.

6.4 Deprecated answeredInRound, Redundant Check

Design Low Version 1 Code Corrected

CS-AGGV2-003

Since Chainlink switched to OCR (Off-Chain Reporting) `answeredInRound` and `roundId` returned by `latestRoundData()` are identical:

https://github.com/smartcontractkit/libocr/blob/c5102a9c0fb776e79cc72d313a61543dc5d48ee2/contract/OffchainAggregator.sol#L883

Chainlinks API Reference for the AggregatorV3Interface describes `answeredInRound` as deprecated:

```
answeredInRound:
Deprecated - Previously used when answers could take multiple rounds to be computed
```

https://docs.chain.link/data-feeds/api-reference#latestrounddata

The corresponding check in `_readChainLink()` is redudant and operates on a deprecated return value (`answeredInRound`):

```
if (
        answeredInRound < roundId
            || (answer <= 0 || uint(answer) > uint(type(uint128).max))
            || updatedAt + ageThreshold < block.timestamp
    )
```

Code corrected:

The redundant check `answeredInRound < roundId` has been removed; these values returned from Chainlink are now ignored.

6.5 Incorrect Check in SetAgreementDistance

Correctness Low Version 1 Code Corrected

CS-AGGV2-009

An authed address can update the agreement distance via `setAgreementDistance()`. Its internal counterparty `_setAgreementDistance()` checks if the new distance (the input) is non-zero, and requires the existing distance (a state variable) to be within _BPS. This makes it possible to set an agreement distance above _BPS and disable the future updates.

```
require(agreementDistance_ != 0);
require(agreementDistance <= _BPS);
```

Code corrected:

Chronicle has corrected the code to enforce the new distance (the input) within _BPS instead of the existing distance (the state variable).

6.6 Gas Optimizations

Informational Version 1 Code Corrected

CS-AGGV2-005

1. The variable `answeredInRound` can be set to `1` instead of `roundId` to save gas, as the compiler does not do that kind of constants propagation.

Code corrected:

The optimization was implemented.

6.7 Meaning of `tieBreakerUsed` Flag

Informational Version 1 Code Corrected

CS-AGGV2-008

The name of the flag `tieBreakerUsed` is ambiguous and has different semantics across the function `_read`:

- In `path=5`, it means that both Chainlink and Chronicle are down and the `TWAP` value is used directly.
- In `path=3`, it means that a tie breaker is used (`_tryTieBreaker()`) due to a large agreement distance between Chainlink and Chronicle.

In general, instead of reflecting if a tie breaker is used, this flag indicates a value independent of that reported by Chainlink and Chronicle has been used in the computation to return a valid price.

**Code corrected:*

The `tieBreakerUsed` flag is no longer present in Version 4.

7 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.

7.1 Age Is `block.timestamp`

Note Version 1

It's important to note that `_age` represents `block.timestamp`. The age of the underlying values can be older, up to the limit defined by the staleness period.

7.2 Immutable Feeds

Note Version 1

All underlying feeds are set as immutable in the constructor and hence these addresses 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.

7.3 Oracle Behavior Under Extreme Circumstances

Note Version 1

If one of the two oracles stops updating, the value reported by Aggor is the value of the working oracle. It's important to understand how the source oracles will behave under extreme circumstances. For example Chainlink paused the update of LUNA. If Chronicle continues to provide values, this value will be reported by Aggor. If Chronicle stops, Aggor will fallback to the Uniswap TWAP (if an Uniswap pool is configured). Upon these changes, the value reported by Aggor may change abruptly.

7.4 Oracles Take Control of Execution

Note Version 1

When Aggor calls external oracles, the control of the execution is transferred to these oracles. If these external calls revert, the execution of Aggor also reverts which breaks the liveness requirement. Aggor could maintain constant control over the execution by wrapping these calls in try-catch blocks. However, correctly implementing this approach can be complex and increases the overhead. Therefore, the decision was made not to use try-catch blocks. Instead, there is an expectation that the external contracts being called, which might be upgradable, will not revert.

In Version 4 the external calls to Chronicle and Chainlink are wrapped in try/catch blocks, allowing Aggor to retain control of the execution flow. A corner case where this doesn't work is described in the comments of the code, namely when returndata decoding fails.

7.5 Understanding the Impact of Using Aggor

Note Version 1

When using an oracle aggregator it's crucial to understand its potential impacts, especially in comparison to relying on a single oracle. Here are some scenarios where Aggor prices may be unreliable:

For Non-Pegged Assets:

- Both Oracles Down + Manipulated Uniswap: If both oracles that Aggor relies on are down, and the Uniswap TWAP is manipulated, Aggor can be manipulated.
- One Oracle Compromised + Manipulated Uniswap: If one of the oracles is under the control of a malicious party, which also manipulates the Uniswap pool, Aggor is compromised (even though the other oracle is working correctly).

For Pegged Assets:

- Unpegging with Only One Oracle Following: If the pegged asset unpegs and only one of the oracles used by Aggor follows this change, there will be discrepancies in the data provided by Aggor.
- Unpegging with Delayed Reaction by One Oracle: In scenarios where an asset unpegs and one oracle reacts slower than the other, Aggor may provide inconsistent data.
- Both Oracles Down + No Uniswap oracle set (expected to be the default case for pegged assets). No value derivation possible.
- Both Oracles Down + Uniswap oracle set but it's manipulated: Similar to non-pegged assets, if both oracles are down and the Uniswap TWAP is manipulated, Aggor can be manipulated.

In summary, while Aggor offers advantages by aggregating data from multiple sources, it's important to recognize situations where its reliability might be compromised.

In Version 4 of the protocol, the oracle functionality *For Pegged Assets* has been removed.

7.6 UniswapV3 Price Manipulation

Note Version 1

If a low liquidity Uniswap pool is used in Aggor, the cost of price manipulation of the internal Uniswap oracle is drastically reduced. While not used as primary oracle, the UniswapV3 price may be used as tie breaker or returned as aggregated price if none of the primary oracle is available. The influence on the security of the aggregator must not be underestimated, the pool used must be resilient against manipulation.