PUBLIC

Code Assessment

of the Price Oracles Smart Contracts

May 15, 2024

Produced for

by

Contents

1 Executive Summary

Dear all,

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

The Notes section highlights important information that users and developers should be aware of.

Euler implements oracle contracts for different providers and a router that maps asset pairs to their corresponding oracle. The contracts are meant to be used in conjunction with Euler's Ethereum Vault Kit (EVK).

The most critical subjects covered in our audit are functional correctness, oracle manipulation resistance and the correctness of protocol integrations.

All contracts show high security in all of the aforementioned subjects after the following issue has been successfully resolved: RedstoneCoreOracle update with stale data.

In summary, we find that the codebase currently 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	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	2
• Code Corrected	1
• Risk Accepted	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 Price Oracles repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	01 May 2024	a89b2a4a89d7806702a27592baad03288c74a633	Initial Version
2	15 May 2024	e6e87993685ca123e5fa542fb784ba5203824769	After Intermediate Report

For the solidity smart contracts, the compiler version `0.8.23` was chosen.

All files in the `./src/` folder were in scope.

2.1.1 Excluded from scope

Any contracts not in `./src/`, in particular the `./lib/` folder is out of scope.

The external systems that the adapters interact with are assumed to function correctly according to their documentation.

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.

Euler offers a library of smart contracts that supports a variety of Price Oracle providers. Each oracle implements the `IPriceOracle` interface, which is designed to abstract the `decimals` of different price feeds from consumers and provide easy composability. Contrary to typical oracle implementations, amounts are converted rather than factors returned.

The system consists of two types of contracts. The `EulerRouter` is an administrative contract that defines which token pairs are supported and routes calls to the correct adapter. Adapter contracts contain the specific implementation for a given oracle type. The EulerRouter essentially acts as a proxy for the adapters. Adapters must be immutable. If an adapter is to be updated, it must be replaced with a new one in the router.

The system will initially be available only on Ethereum mainnet, but deployments on other EVM-compatible chains will be considered in the future.

2.2.1 IPriceOracle

The `IPriceOracle` interface contains the following functions:

- `name()`: Returns the name of the oracle.
- `getQuote()`: Returns the `outAmount` of `quote` token that would be received for trading `inAmount` of `base` token, assuming no price spread.
- `getQuotes()`: Returns `bidOutAmount` and `askOutAmount`, which are the amounts that would be received for trading when taking the price spread into account.

The following must always hold: `bidOutAmount <= outAmount <= askOutAmount`. However, anything beyond this is not guaranteed. In fact, all oracles in scope of this review inherit the BaseAdapter, which is a simplified implementation of the IPriceOracle interface that always returns the same price for `getQuote()` and both values of `getQuotes()`.

In the future, more advanced oracles can be added that take into account factors such as market depth. This can be useful for valuing collateral, as the execution price of a large trade can be significantly different from the price of a small trade.

2.2.2 EulerRouter

The `EulerRouter` implements the `IPriceOracle` interface and contains the following functions:

Governor functions:

- `govSetConfig()`: Sets the adapter for a given token pair.
- `govSetResolvedVault()`: Sets an ERC4626 vault that is trusted to correctly convert its shares to underlying tokens using the `convertToAssets()` function.
- `govSetFallbackOracle()`: Sets the fallback oracle that is used in case a pair has no oracle configured.

User functions:

- `getQuote()`: Resolves and calls the configured adapter for a given token pair and returns the result.
- `getQuotes()`: Resolves and calls the configured adapter for a given token pair and returns the result.
- `getConfiguredOracle()`: Returns the configured oracle for a given token pair.
- `resolveOracle()`: Returns the oracle for a given token pair, but first recursively resolves the `base` token from ERC4626 shares to underlying tokens.

The Governor is trusted to only add suitable adapters to the router. For resolved vaults, it must ensure that the `convertToAssets()` function is not manipulatable, e.g. through donation attacks.

2.2.3 BaseAdapter

BaseAdapter is the base contract for all oracle adapters currently in scope. It implements the `IPriceOracle` interface. It additionally contains logic that allows child contracts to fetch decimals from ERC-20 assets. In case an asset address does not expose a `decimals()` function, a default of 18 decimals is returned. This is used to support non-ERC20 assets like, for example, ERC-7535 or off-chain assets.

2.2.4 CrossAdapter

The `CrossAdapter` is an adapter that chains two adapters together. Given two tokens X and Y and an intermediate token Z, the CrossAdapter first calls an oracle converting X to Z, then calls an oracle converting Z to Y. This is useful for converting tokens that do not have a direct price feed.

2.2.5 ChainlinkOracle

`ChainlinkOracle` is an adapter for Chainlink price feeds. It supports exactly one price feed and always reports the latest price.

A `maxStaleness` threshold is configured upon deployment. If the latest price is older than this threshold, the call will revert.

2.2.6 ChronicleOracle

`ChronicleOracle` is an adapter for Chronicle price feeds. It supports exactly one price feed and always reports the latest price.

A `maxStaleness` threshold is configured upon deployment. If the latest price is older than this threshold, the call will revert.

2.2.7 PythOracle

`PythOracle` is an adapter for Pyth price feeds. It supports exactly one price feed and always reports the latest price.

A `maxStaleness` threshold is configured upon deployment. If the latest price is older than this threshold, the call will revert. Additionally, the price may not be more than one minute ahead of the `block.timestamp`.

There is also a `maxConfWidth` parameter configured upon deployment. The confidence interval returned by Pyth is an indication of how much the price sources agree on the price. If the confidence interval is wider than `maxConfWidth`, the call will revert.

Pyth is a pull-based system. This means that the oracle must be called by the consumer to update the price. It is expected that consumers will call `updatePriceFeeds()` (on the Pyth contract) and `getQuote()` in the same transaction, if the old price has become stale. This can be done using a multicall contract such as the Ethereum Vault Connector (EVC).

2.2.8 RedstoneCoreOracle

`RedstoneCoreOracle` is an adapter for RedStone price feeds.

It supports exactly one price feed and always reports the latest price.

A `maxStaleness` threshold is configured upon deployment. If the latest price is older than this threshold, the call will revert. Additionally, the price may not be more than one minute ahead of the `block.timestamp`.

Redstone is a pull-based system. This means that the oracle must be called by the consumer to update the price. It is expected that consumers will call `updatePrice()` and `getQuote()` in the same transaction, if the old price has become stale. This can be done using a multicall contract such as the Ethereum Vault Connector (EVC).

The `RedstoneCoreOracle` inherits the `PrimaryProdDataServiceConsumerBase` contract from the RedStone library. This contract has the RedStone PrimaryProd signers hardcoded, alongside a 3/5 signing threshold. If the signers or threshold need to be changed, a new contract must be deployed that overwrites these values. This new contract would then be configured in the EulerRouter.

2.2.9 UniswapV3Oracle

The `UniswapV3Oracle` is an adapter for Uniswap V3 TWAPs.

It supports exactly one pool and reports the TWAP price.

The `twapWindow` (in seconds) is configured on deployment. It must be ensured that the cardinality of the observation buffer of the pool has been expanded enough to cover the `twapWindow`. Additionally, the pool should have sufficient liquidity, including full-range liquidity, to be difficult to manipulate.

2.2.10 LidoOracle

LidoOracle uses the `stETH` token's `getSharesByPooledEth()` and `getPooledEthByShares()` functions to convert between `stETH` and `wstETH`.

The adapter can only be used on Ethereum mainnet, as the canonical `stETH` contract is only deployed there.

2.2.11 Roles & Trust Model

The `EulerRouter` admin role (Governor) is fully trusted, as it can replace any oracle implementation. In the worst case, it could replace any oracle with a contract that returns an arbitrary hardcoded value.

Protocol developers may choose to deploy their own version of `EulerRouter` (and any of the oracle contracts).

Oracles are deployed immutably and replaced in the router in case they have to be changed.

The adapters inherit the trust model of the oracle system that they are calling.

We assume that the oracles are deployed with well-chosen configurations as they are highly relevant for the security of the contracts.

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:

- 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	1

5.1 Missing Min/Max Checks in ChainlinkOracle

Security Low Version 1 Risk Accepted

CS-EULPO-002

Chainlink aggregators have a minimum and maximum price defined. Any updates to the aggregators are rejected if these thresholds are crossed. In most cases, this is not problematic. There are, however, certain feeds for which the thresholds are defined in a way that could be reached in certain market events.

If the thresholds are crossed, the oracle prices won't update anymore. Since most Chainlink feeds are configured with rather high heartbeats (e.g., 24 hours), it takes some time until this can be reliably detected by the `ChainlinkOracle`.

In the rare case this happens, it can have catastrophic effects on any protocol relying on the oracle, as stale prices will be reported.

Risk accepted:

Euler accepts the risk with the following statement:

> We have decided not to include `minAnswer` and `maxAnswer` bounds to `ChainlinkOracle`. It is unclear whether we can expect these value to change so we cannot safely store them as `immutable` variables. We expect `ChainlinkOracle` to be a very hot contract for markets and we believe the gas cost increase associated with runtime fetching of `minAnswer` and `maxAnswer` is not worth it. Ultimately we believe these values are chosen responsibly by the Chainlink team. This opinion is evidently shared by the largest users of Chainlink, which also do not have `minAnswer` and `maxAnswer` checks.

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	0
Medium-Severity Findings	1

- RedstoneCoreOracle Update With Stale Data Code Corrected

Low-Severity Findings	1

- RedstoneCoreOracle DoS Code Corrected

Informational Findings	3

- Typographical Error Code Corrected
- RedstoneCoreOracle Forced Package Ordering Code Corrected
- Missing Uniswap Oracle Documentation Specification Changed

6.1 RedstoneCoreOracle Update With Stale Data

Correctness Medium Version 1 Code Corrected

CS-EULPO-001

`RedstoneCoreOracle` implements the `PrimaryProdDataServiceConsumerBase` of Redstone's SDK. This contract manages the validation of submitted calldata to the `updatePrice()` function. Such calldata contains a minimum of three different packages that each contain a price for the requested feed, as well as a timestamp (and the signature of the associated Redstone signer). Timestamps, after being extracted from calldata, are handled in the function `validateTimestamp()` which checks their staleness. This is done individually for each submitted package.

`RedstoneCoreOracle` overrides the `validateTimestamp()` function to add custom staleness checks as well as a storage write of the given timestamp. To avoid redundant writes, the function returns early when a given timestamp is equal to the timestamp that has been previously written to storage:

```
function _validateTimestamp(uint256 timestampMillis) internal {
    // The `updatePriceContext` guard effectively blocks external / direct calls to `validateTimestamp`.
    Cache memory _cache = cache;
    if (_cache.updatePriceContext != FLAG_UPDATE_PRICE_ENTERED) revert Errors.PriceOracle_InvalidAnswer();

    uint256 timestamp = timestampMillis / 1000;
    // Avoid redundant storage writes as `validateTimestamp` is called for every signer in the payload (3 times).
    // The inherited Redstone consumer contract enforces that the timestamps are the same for all signers.
    if (timestamp == _cache.priceTimestamp) return;

    if (block.timestamp > timestamp) {
        // Verify that the timestamp is not too stale.
        uint256 priceStaleness = block.timestamp - timestamp;
```

```
        if (priceStaleness > maxStaleness) {
            revert Errors.PriceOracle_TooStale(priceStaleness, maxStaleness);
        }
```

```
    } else if (timestamp - block.timestamp > RedstoneDefaultsLib.DEFAULT_MAX_DATA_TIMESTAMP_AHEAD_SECONDS) {
        // Verify that the timestamp is not too long in the future (1 min). Redstone SDK explicitly allows this.
        revert Errors.PriceOracle_InvalidAnswer();
    }

    // Enforce that cached price updates have a monotonically increasing timestamp.
    if (timestamp < _cache.priceTimestamp) revert Errors.PriceOracle_InvalidAnswer();
    cache.priceTimestamp = uint48(timestamp);
}
```

This is done under the assumption that the timestamps on all submitted packages are equal. `PrimaryProdDataServiceConsumerBase`, however, never enforces this, leading to the issue depicted by the following example:

1. `updatePrice()` is called with three packages containing the prices [1, 1, 1] and timestamps [1, 1, 1].
2. The cached price is set to 1, as is the cached timestamp.
3. After some time, the price of the respective asset has changed a lot. A new price update would contain three packages with the prices [2, 2, 2] and timestamps [2,2, 2].
4. A malicious user crafts calldata that contains two of the old prices and timestamps ([1, 1, 2] and [1, 1, 2]) and calls `updatePrice()` with it.
5. `validateTimestamp()`, for the first two packages, returns early because the timestamps are equal to the old timestamp in the cache. For the third package, it performs staleness checks and writes the new timestamp 2 into the cache.
6. `updatePrice()` writes the median of the received prices ([1, 1, 2] -> 1) into the cache. The oracle now contains the price 1 with a timestamp of 2.
7. The user can now call `getQuote()` to retrieve the price without revert as the price is not stale (assuming that the price would be stale with timestamp 1).

The longer the oracle has not been updated, the more severe this problem becomes, as the price can always be kept at the price of the last update while the timestamp is reset to a current one. The fact that prices are aggregated by median value exacerbate this problem further.

6.2 RedstoneCoreOracle DoS

Design | Low | Version 1 | Code Corrected

CS-EULPO-008

`RedstoneCoreOracle.updatePrice()` enforces monotonically increasing timestamps. If a user updates the price with a timestamp that is lower than the one stored in the cache, the transaction reverts.

Since price signatures are published rapidly, it is possible that two users submit price updates at roughly the same time but with different timestamps. If the transaction of the user with the lower timestamp is executed last, it will revert. Users should be aware that even if their price update transaction fails, there may still be a valid price to read afterwards.

Consider the following example:

1. User Alice submits a multicall transaction that updates the price and then reads it.
2. Alice is frontrun by someone else who updates the price to an even newer value.
3. The `updatePrice()` call reverts.
4. As the transaction has reverted, the rest of Alice's transaction will also revert even though it could have used the more up-to-date price.

Protocol developers should consider catching reverts in price updates when executing them as part of a multicall. Otherwise unnecessary reverts could result in Denial of Service.

Code corrected:

`RedstoneCoreOracle.updatePrice()` no longer reverts when a price update with an outdated timestamp occurs. It is worth to note that this now means that users' transactions might now be executed with a different price than anticipated.

6.3 Missing Uniswap Oracle Documentation

Informational | Version 1 | Specification Changed

CS-EULPO-004

The documentation of `UniswapV3Oracle` contains a warning detailing in which circumstances the oracle can be used (mostly) safely. Among other things, the cardinality of the observation buffer and the need for enough liquidity is mentioned. While this is correct, it could be further extended:

1. The buffer must not only have enough cardinality, but there also must have been enough observations since it was extended to fully fill it.
2. Not only the liquidity at the current time is relevant, but also the liquidity while the buffer was filled with data.

Specification changed:

`UniswapV3Oracle` now contains appropriate documentation to make users aware of the additional requirements.

6.4 RedstoneCoreOracle Forced Package Ordering

Informational | Version 1 | Code Corrected

CS-EULPO-005

Since `RedstoneCoreOracle.validateTimestamp()` enforces monotonically increasing timestamps, packages with differing timestamps must be ordered in the calldata so that the call does not revert.

This is different to the regular Redstone SDK implementation and may lead to incompatibilities. Developers should be aware of this restriction.

`RedstoneCoreOracle` now requires timestamps of all data packages to be equal.

6.5 Typographical Error

Informational | Version 1 | Code Corrected

CS-EULPO-007

The comments of `RedstoneCoreOracle.updatePrice()` contain the following line which is missing a comma: "*During execution the context flag is set to `FLAG_UPDATE_PRICE_ENTERED`*.".

Code corrected:

The comment has been removed as the respective functionality is no longer present.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 L2 Sequencer Considerations

Informational **Version 1** **Acknowledged**

CS-EULPO-003

The contracts will initially be deployed to Ethereum mainnet only but should eventually work on any L2.

Before deploying to L2s, consider that the L2 sequencer can have extended downtime.

The Chainlink docs suggest using the SequencerUptimeFeed to detect this and not consume any prices until the sequencer is back up.

Also, the Uniswap TWAP will have a high weight for the price that was present during the downtime (as it persisted for a long time). This could be a problem if the price was an outlier or if the price when the sequencer recovers is significantly different from when it went down.

Additionally, `block.timestamp` may have slightly different behavior on other chains. This should be considered before deploying.

Acknowledged:

Euler acknowledges the risk associated with the mentioned markets.

7.2 Redundant Chronicle Adapter

Informational **Version 1** **Acknowledged**

CS-EULPO-006

The `ChronicleOracle` communicates with Chronicle price feeds using the function `readWithAge()`. Chronicle, however, also exposes the function `latestRoundData()` that is equivalent to Chainlink's feed interface. Since only positive prices are used, the `ChainlinkOracle` should therefore be compatible with Chronicle as well.

Acknowledged:

Euler acknowledges that the contract is redundant.

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 Compounding Errors in CrossAdapter

Note Version 1

`CrossAdapter` allows chaining oracle calls in order to get price quotes for pairs with no direct feed available. Users should be aware that using such an oracle might increase the margin of error in comparison to using a single oracle. This is especially relevant for pull-based oracles that give users the ability to "choose" their price in a narrow margin, but also applies to push-based oracles.

Consider the following example:

- A CrossAdapter chains two instances of PythOracle
- Both oracles have an outdated price (that is not stale yet), which is 0.5% different from the newest price
- Usually, a user would be able to choose between the old price and the new price, which gives them a 0.5% choice.
- In this case, the user can make this choice twice, so in the worst case, they may be able to use a price that is ~1% different from the current price.

Any errors in chained oracles will be accumulated when using CrossAdapter. The resulting wider error margin should be included in risk assessments by consumers.

8.2 Lido Oracle Staleness

Note Version 1

The `LidoOracle` converts stETH to wstETH and vice versa. This is done by calling the functions `getSharesByPooledEth()` and `getPooledEthByShares()` respectively on the stETH contract.

The pooled ETH value in Lido is only updated periodically.

There are two main cases where the ETH per share will change:

1. Staking rewards accrue (small increase)
2. Validators are slashed (variable size decrease)

Both of these changes are possible to predict before they are reflected on-chain. Someone could frontrun the update and use the stale price, with knowledge of what the future price will be.

Protocols using the `LidoOracle` must take this into account and should especially consider the slashing case. A slashing event will only be reflected by `LidoOracle` long after it has happened.

8.3 No Price Quote in Vault Tokens

Note Version 1

`EulerRouter.resolveOracle()` allows to find the configured oracle for a given asset pair. If the `base` token is the token of an ERC-4626 vault and the vault has been configured in the router, the corresponding oracle can be resolved by first converting the vault's token to the corresponding amount of the vault's asset and then solving for an oracle of the vault's asset.

Note that this is not supported for the `quote` token. Prices cannot be quoted in vault tokens even if the corresponding vault is be configured in the router.

8.4 Oracle Consumers Must Handle Reverts

Note **Version 1**

Most of the implemented oracle adapters have certain conditions that can cause reverts. One reason could be a price becoming stale.

Oracle consumers must be aware of this and should mitigate the impact of a reverting oracle as much as possible. Depending on design, a revert in the oracle may cause an unnecessary denial of service to the whole system.

For example, it should be considered if it is possible to allow withdrawals even when an oracle is down.

8.5 Oracle Manipulation on FIFO L2s

Note **Version 1**

Price oracles must be robust and manipulation-resistant. It must be expensive to manipulate the markets which are used as price sources.

Usually, the factor that makes price manipulation expensive is arbitrage. If a manipulator pushes the price of an asset too high or too low, arbitrageurs will see this and make a profit by moving the price back to the "true price". Any profit made by arbitrageurs will be a loss to the manipulator.

This "arbitrage assumption" breaks down in two cases:

1. All markets for the token are manipulated simultaneously, so it is difficult to determine the "true price". There is no other market to arbitrage against.
2. Arbitrageurs are not able to see the manipulated price quickly enough, so they cannot take advantage of it.

Attacks that target condition 2. are known as "Multi-block MEV" attacks. The idea is that a manipulator could control the order of transactions in a block, which allows the following:

- In block `n`, the manipulator sends a transaction (through a private mempool like Flashbots) that manipulates the price of an asset.
- In block `n+1`, the manipulator ensures that the first transaction in the block is one where they revert the price back to the original value.

As a result, arbitrageurs will have no chance of reacting to the manipulation, as it will already be over by the time they can get a transaction included in the block. However, if there is an Oracle that reads the price at the beginning of block `n+1`, it will see the manipulated price.

This attack is well-known on Ethereum, but is generally deemed expensive to execute, as it requires being or having an agreement with the ETH staker that is chosen to propose block `n+1`. If the attack should be repeated multiple times, it requires being chosen as block proposer multiple times within a short time frame, which requires a significant amount of ETH staked.

However, on L2s, block production works differently. Instead of a different proposer being chosen in each block, there is typically a single sequencer that decides on a block ordering policy. One commonly used policy used by chains such as Arbitrum, Optimism and Base is "FIFO" (First In, First Out), where transactions are included in the order they were received.

In FIFO ordering, the order of transactions is determined by time, not by the price a user is willing to pay. This can be taken advantage of to fulfill condition 2. above, without needing to be a block producer.

The FIFO attack looks as follows:

1. The manipulator experiments to figure out their latency to the sequencer (and ideally minimizes it).
2. The manipulator sends a manipulation transaction at a time such that it will arrive at the sequencer towards the end of the period in which it is building block `n`.
3. The manipulator sends a second transaction so that it reaches the sequencer at the beginning of the period in which it is building block `n+1`.

Arbitrageurs are only able to see the manipulated price once block `n` is published by the sequencer. By that time, the manipulator has already sent the second transaction that reverts the price back to the original value. As time is the only relevant factor, it is impossible for a transaction that is created later to be included in the block first (unless the arbitrageur has significantly lower latency to the sequencer).

The only cost to the attacker is the trading fees paid. As the attack cannot use a flashloan, they must also have sufficient capital available to manipulate the price by the percentage they aim for. The attack can be repeated as many times as the attacker wants, although repeated attacks could be speculatively frontrun by arbitrageurs if they detect a pattern. Repeated attacks can be used to circumvent outlier-detection mechanisms and TWAPs.

A policy that modifies transaction ordering to be based on a payment in addition to timing would make the attack significantly more expensive. For example, "Arbitrum time-boost" has been proposed, but not yet implemented. See Time Boost Medium post.

Note that Multi-block MEV attacks have historically been considered mostly in the context of TWAP manipulation. However, if there is an off-chain oracle, such as ChainLink, that uses an on-chain market as a primary price source, the attack also applies there. In fact, the effect will be much larger, as off-chain oracles typically do not use a time-weighted average. Instead, they read the spot price at a single point in time. As a result, executing the attack once could lead to a heavily manipulated price. Some off-chain oracles may implement outlier-detection to mitigate this, but this is often not clearly documented, if it exists at all. If outlier-detection exists, the attack could be executed multiple times.

In summary, Multi-block MEV attacks are likely much more realistic to execute on FIFO L2s than on Ethereum, as they are possible without needing to be a block producer. They can affect on-chain TWAPs as well as any off-chain oracles that use L2 on-chain markets as a primary price source. This must be considered when deciding which assets have an oracle that is robust enough to allow lending.

8.6 Potentially Incorrect Default Decimals

Note **Version 1**

`BaseAdapter` (and therefore all oracle adapters in the version of the protocol at the time of this report) fetches decimals of the underlying base and quote assets by using the function `_getDecimals()`:

```
function _getDecimals(address asset) internal view returns (uint8) {
    (bool success, bytes memory data) = asset.staticcall(abi.encodeCall(IERC20.decimals, ()));
    return success && data.length == 32 ? abi.decode(data, (uint8)) : 18;
}
```

The function attempts to fetch the decimals of a given token by calling the associated `decimals()` function. If this call fails, it instead uses a default of 18 decimals.

This behavior is meant to support, among others, asset types that are not smart contracts (e.g., the native asset address `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`). There is, however, a certain caveat that should be considered before deploying an oracle with such addresses:

> Deployment of an oracle before the actual ERC-20 asset is deployed with `CREATE2` can result in wrong decimals.

Users should avoid using oracle adapters that are misconfigured in this fashion.

8.7 Pyth Maximum Confidence

Note **Version 1**

The `PythOracle` allows to set a `maxConfWidth` interval that defines the maximum confidence the oracle is allowed to have before reverting. Note that this interval should be initialized with a value that also covers possibly wider margins than can be usually observed. This is due to the fact that the confidence interval naturally widens in more volatile market conditions, which are exactly the market conditions in which liquidations should occur in a timely manner. If the threshold prevents swift liquidations in such a case, it could be counter-productive.

8.8 RedstoneCoreOracle Hot-Swap

Note **Version 1**

`RedstoneCoreOracle` contains constant signer addresses. Should there be any complications with these signer addresses, the oracle has to be swapped for a new contract that contains an updated set of signers. Since the contract is immutable (i.e., not deployed behind a proxy) it is important that any projects using it implement functionality that allows replacement of the oracle.