SPEARBIT

Euler Labs - Euler Price Oracle Security Review

Auditors

Christoph Michel, Lead Security Researcher

Emanuele Ricci, Lead Security Researcher

M4rio.eth, Security Researcher

Christos Pap, Associate Security Researcher

David Chaparro, Junior Security Researcher

Report prepared by: Lucas Goiriz

May 20, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Euler Labs is a team of developers and quantitative analysts building DeFi applications for the future of finance.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of euler-price-oracle according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 30 days in total, Euler Labs engaged with Spearbit to review the euler-price-oracle protocol. In this period of time a total of **36** issues were found.

Summary

Project Name	Euler Labs
Repository	euler-price-oracle
Commit	eeb184...1a65e6
Type of Project	DeFi
Audit Timeline	Apr 8 to May 10
Two week fix period	May 10 - May 20

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	2	0
Low Risk	4	2	2
Gas Optimizations	1	1	0
Informational	29	14	15
Total	**36**	**19**	**17**

5 Findings

5.1 Medium Risk

5.1.1 `RedstoneCoreOracle.updatePrice` can revert and use stale prices as not all price timestamps need to be the same

Severity: Medium Risk

Context: RedstoneCoreOracle.sol#L121-L122

Description: The Redstone oracle allows users to provide signed data packages through the `updatePrice` function (signed by verified parties with a minimum threshold of signers required). The current code operates on the assumption that all data packages contain the same timestamp. However, the Redstone SDK does not check this for the `getOracleNumericValueFromTxMsg` entrypoint.

```
// the following is not true
// The inherited Redstone consumer contract enforces that the timestamps are the same for all signers.
if (timestamp == _cache.priceTimestamp) return;
```

As the code checks that all timestamps are increasing the cached timestamp, unordered data packages will revert.

There's a second bug that attackers can inject old data packages into `updatePrice` that use the old cached timestamp (which might already be stale) and the early return will skip validation for these, but they are still used to compute the median price value.

Recommendation: Remove the comment and the early return. There is no single timestamp for an aggregated Redstone value. There are several approaches with different tradeoffs:

- taking the maximum/average timestamp, checking it is greater than the cached one: two almost-stale prices can be matched with a fresh price. Enforcing a "monotonically increasing timestamp" does not say much about the freshness of the price compared to the previous price.
- taking the maximum/average timestamp, all individual data package timestamps are checked to be greater than the cached one: fewer data packages can be used to update a price. the heartbeat of the signers must be observed so the cached prices don't go stale without being able to update them.

Euler: Fixed in PR 40. All the packages are now forced to have the same timestamp as the one provided by the `updatePrice` called.

Spearbit: Verified. Euler should note and be aware the current implementation of `validateTimestamp` relies on the assumption that all the signed prices provided in the `updatePrice` calldata have the same `timestamp` (information that is attached to the specific single price itself and not directly to the aggregated data). If this `Redstone` logic changes in the future, and each price has a different `timestamp` (that diff off by more than a second from the `updatePrice` parameter), the `RedstoneCoreOracle` will revert inside `validateTimestamp`, failing to update the price.

5.1.2 `RedstoneCoreOracle` could allow picking up new price that are older compared to the current one

Severity: Medium Risk

Context: RedstoneCoreOracle.sol#L70-L93

Description: The current logic of the `RedstoneCoreOracle` implementation uses `block.timestamp` as the timestamp to which a price has been observed instead of the **real** observation timestamp that is bound to the price itself. The `validateTimestamp` function allows the user to pick up a price that is at most `maxPriceStaleness` seconds in the past and one minute in the future. The comparison is made between the oracle price observed timestamp and `block.timestamp` but is not verifying that the price's timestamp is older than the current one stored in the contract inside the `cacheUpdatedAt` and `cachedPrice`.

Given the checks made in `updatePrice`, `validateTimestamp` and `_getQuote` it's possible for an attacker to pick up a price in the several data packages that will be considered valid and will be older than the current selected one.

This could allow an attacker to pick a price that will benefit the operation performed on vaults where such price will be used, borrowing more than expected or liquidating a user that should not be liquidated.

Let's assume we are in the current configuration:

- `maxCacheStaleness = 60` 1 minute.
- `maxPriceStaleness = 180` 3 minutes, this is the one used by default by Redstone in their SDK, see `RedstoneDefaultsLib`.

Starting data:

- `cachedPrice = 0`
- `cacheUpdatedAt = 0`
- `block.timestamp = 1000`

Flow:

- T0: updatePrice() is called with a `price.timestamp` (signed price timestamp) 60 seconds in the future compared to `block.timestamp`.
  - `price.timestamp = 1060`
  - `updatePrice` check → `block.timestamp <= maxCacheStaleness + cacheUpdatedAt` → `1000 <= 60+0` → `FALSE` → do not revert, OK.
  - `validateTimestamp` check → `timestamp - block.timestamp > TIMESTAMP_AHEAD_SECONDS` → `1060 - 1000 > 180` → FALSE → do not revert, OK.

  All the checks passed meaning that:

  - `cacheUpdatedAt` is updated at 1000
  - `cachedPrice` is updated at X (price1.value)
- T0+: At this point, any call to `updatePrice()` done before `cacheUpdatedAt + maxCacheStaleness + 1` will skip the price update.
- T1: `block.timestamp = 1061**`, call `updatePrice` with `newPrice.timestamp = 1059`

`updatePrice` check → `block.timestamp <= maxCacheStaleness + cacheUpdatedAt` → `1061 <= 60+1000` → `FALSE` → do not revert, OK. `validateTimestamp` check: - → `staleness = block.timestamp - timestamp = 1061-1059 = 2` - → `staleness > maxPriceStaleness = 2 > 180` → `FALSE` → do not revert, OK

- `cacheUpdatedAt` is updated at `1061`
- `cachedPrice` is updated at X (`price2.value`)

The result is that `price2` was older than `price1` but the `RedstoneCoreOracle` has allowed the caller to submit and store such a price (`price1.timestamp = 1060`, `price2.timestamp - 1059`).

Test: The following test replicates the behavior of the `RedstoneCoreOracle` modifying the logic of the contract for the sake of simplicity:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity 0.8.23;

import {Test, console} from "forge-std/Test.sol";
import {Errors} from "src/lib/Errors.sol";

contract RSNewPriceInThePastTest is Test {
    uint256 DEFAULT_MAX_DATA_TIMESTAMP_AHEAD_SECONDS;
    uint256 maxPriceStaleness;
    uint256 maxCacheStaleness;

    uint208 cachedPrice;
    uint48 cacheUpdatedAt;
    uint256 cachedId;

    struct RedstonePrice {
        uint256 id;
        uint48 timestamp;
        uint208 price;
    }

    function setUp() public {
        DEFAULT_MAX_DATA_TIMESTAMP_AHEAD_SECONDS = 60; // the same as the one from Redstone
    }

    function testPriceInThePast() external {
        maxCacheStaleness = 60;
        maxPriceStaleness = 180;

        // set the block.timestamp to 1000
        vm.warp(1000);
        assertEq(block.timestamp, 1000);

        /////////////
        // FIRST PRICE: in the future of 60 seconds
        /////////////

        // propose first price, it will be in the future of 60 seconds (max allowed)
        uint48 price1Timestamp = 1060;
        RedstonePrice memory priceInTheFuture = RedstonePrice({id: 1, timestamp: price1Timestamp *
↪   1000, price: 1000});
        updatePrice(priceInTheFuture);

        assertEq(cachedId, priceInTheFuture.id);
        assertEq(cachedPrice, priceInTheFuture.price);
        assertEq(cacheUpdatedAt, 1000); // current block.timestamp

        // just assert that it does not revert
        _getQuote(1, address(0), address(0));

        /////////////
        // SECOND PRICE: 1 second BEFORE the current price signed timestamp
        /////////////

        // we want now prove that we are able to make the oracle accept a price that was in the PAST
        // compared to the price.timestamp (the signed timestamp of the price itself)
        // note: prev price timestamp was 1060, this price timestamp is 1059
        uint48 price2Timestamp = price1Timestamp - 1;
        RedstonePrice memory priceInThePast = RedstonePrice({id: 2, timestamp: price2Timestamp * 1000,
↪   price: 500});

        // if we try to update before that `maxCacheStaleness` has passed, it will just return and skip
↪   it
        vm.warp(1000 + maxCacheStaleness); // still need 1 second
        assertEq(block.timestamp, 1000 + maxCacheStaleness);

        // just assert that it does not revert
        updatePrice(priceInThePast);

        // assert that nothing has changed
        assertEq(cachedId, priceInTheFuture.id);

        // warp 1 second more compared to before, now we should be able to
        vm.warp(1000 + maxCacheStaleness + 1);
        assertEq(block.timestamp, 1000 + maxCacheStaleness + 1);
        // now we can update the price
        updatePrice(priceInThePast);

        assertEq(cachedId, priceInThePast.id);
        assertEq(cachedPrice, priceInThePast.price);
        assertEq(cacheUpdatedAt, 1000 + maxCacheStaleness + 1); // current block.timestamp

        // just assert that it does not revert
        _getQuote(1, address(0), address(0));
    }

    function updatePrice(RedstonePrice memory proposedPrice) internal {
        // Use the cache if it has not expired.
        if (block.timestamp <= maxCacheStaleness + cacheUpdatedAt) return;
        proposedPrice = getOracleNumericValueFromTxMsg(proposedPrice);
        if (proposedPrice.price > type(uint208).max) revert Errors.PriceOracle_Overflow();

        // update values
        cachedId = uint48(proposedPrice.id);
        cachedPrice = uint208(proposedPrice.price);
        cacheUpdatedAt = uint48(block.timestamp);
    }

    function getOracleNumericValueFromTxMsg(RedstonePrice memory proposedPrice)
        internal
        returns (RedstonePrice memory)
    {
        // redstone validation here
        //
        // callback to validate the price timestamp
        //
        validateTimestamp(proposedPrice);

        // validation performed, return the validated price
        return proposedPrice;
    }

    function validateTimestamp(RedstonePrice memory proposedPrice) internal {
        uint256 timestamp = proposedPrice.timestamp / 1000;
        if (block.timestamp > timestamp) {
            uint256 staleness = block.timestamp - timestamp;
            console.log("block.timestamp", block.timestamp);
            console.log("timestamp", timestamp);
            console.log("staleness", staleness);
            console.log("maxPriceStaleness", maxPriceStaleness);
            if (staleness > maxPriceStaleness) revert Errors.PriceOracle_TooStale(staleness,
↪   maxPriceStaleness);
        } else if (timestamp - block.timestamp > DEFAULT_MAX_DATA_TIMESTAMP_AHEAD_SECONDS) {
            revert Errors.PriceOracle_InvalidAnswer();
        }
    }

    function _getQuote(uint256 inAmount, address _base, address _quote) internal view returns (uint256)
↪   {
        uint256 staleness = block.timestamp - cacheUpdatedAt;
        if (staleness > maxCacheStaleness) revert Errors.PriceOracle_TooStale(staleness,
↪   maxCacheStaleness);

        // we don't care, it just need not revert
        return 1;
    }
```

```
}
```

Recommendation: Given that Redstone oracle offers "observation timestamp" for their prices, Euler should take advantage of such feature and store such timestamp in the `cacheUpdatedAt` state variable instead of `block.timestamp`. Any price that is before the observation timestamp of the current cached price should be considered invalid.

Unfortunately, it appears that the Redstone SDK does not offer an easy way to override the needed functions or inject callbacks/hooks to pick the observation timestamp of the selected price returned by `getOracleNumericValueFromTxMsg`.

Euler should:

- Document this behavior and pick `maxCacheStaleness` and `maxPriceStaleness` to reduce as much as possible this scenario.
- Rebuild the Redstone SDK in a way that allows them to pick the `extractedTimestamp` of the selected price returned by `getOracleNumericValueFromTxMsg`.

Euler: The issue has been mitigated by rewriting the RedStone Oracle to a more lightweight version in commit 5c5cbf8d.

Spearbit: Resolved.

5.2 Low Risk

5.2.1 `BaseAdapter._getDecimals` returns 18 decimals when the `asset` contract does not implement the `decimals` function

Severity: Low Risk

Context: BaseAdapter.sol#L29-L32

Description: When the `asset` contract passed as a parameter is a contract that does not offer the `decimals` function, the `staticcall` function will fail and return `18` as the asset's decimal.

This behavior could lead to a wrong assumption that could be dangerous if used in a price conversion. In addition to this problem, the `_getDecimals` has the following edge case that should be considered:

- EOA does not revert and will return `18`.
- non-deployed contracts (`asset.code.length == 0`) does not revert and will return `18`.
- Contracts that do not implement `decimals` do not revert and will return `18`.
- Contracts that implement `decimals` but with a different return type (let's say `uint256`) do not revert if the returned value is `<= type(uint8).max` and will return the returned value.

Recommendation: Euler should document all these edge cases and ensure that the `asset` passed as a parameter to the `_getDecimals` is indeed a valid ERC20 compliant contract.

Euler: Fixed in commit 8d16b859. We want oracle adapters to be able to represent non-crypto assets. The concrete reason now is USD-denominated pairs. `address(840)` has 18 decimals according to `_getDecimals`. But this also allows us to represent other ISO-4217 currencies without special-casing them in `_getDecimals`.

We also have the flexibility to apply other conventions to represent other asset classes as 160-bit values.

Spearbit: Verified.

5.2.2 The `PythOracle` won't work with positive `exp` parameter

Severity: Low Risk

Context: PythOracle.sol#L82

Description: The `Pyth` oracle is using an `exp` parameter to determine the decimals that the `price` will be normalized to result in the actual price. The formula is: `price * 10^exp`.

We checked all the current feeds and the `exp` value is between -5/-10. The current Solidity SDK also states that the `exp` can **NOT** be `>= 0`.

While we agree with most of the aforementioned reasonings to only support negative values for the `exp`, we've noticed in the code of the Pyth Client that the actual `exp` value can be negative and positive:

- add_price.rs#L44: When you add a price, it checks the exponent.
- utils.rs#L101-L106: It can be between `+-MAX_NUM_DECIMALS`
- c_oracle_header.rs#L14: The `MAX_NUM_DECIMALS` has a value of 12 so theoretically it can be `+-12`.

Furthermore, we've talked with the `Pyth` team and they confirmed that currently they have set the check in the SDK to facilitate the discussions but they do not exclude the fact that this value can be positive in the future.

Recommendation: Consider supporting positive `exp` for a more generalized integration of the `PythOracle`.

Euler: Resolved in PR 32.

Spearbit: Verified.

5.2.3 Chainlink price feed decimals are cached

Severity: Low Risk

Context: ChainlinkOracle.sol#L39-L40, ETHUSD price feed, Chainlink feeds

Description: The Chainlink adapter fetches the Chainlink price feed's `decimals` once in the constructor. As Chainlink price feeds fetch the decimals from the *current aggregator* and the aggregator can be changed, the `decimals` could also change which would lead to wrong price conversion if the price with new decimals is used with the old cached `decimals`. It's unclear if Chainlink would ever update the aggregator to one with different `decimals`.

Recommendation: Ideally, the `decimals` would be fetched every time the `latestRoundData` function is called to get the price.

Euler: Acknowledged, won't fix. We discussed this at length and won't be applying the recommendation due to the following reasons:

- There is no past record of Chainlink changing the decimals of an already live price feed.
- Chainlink oracles secure billions of DeFi value, changing decimals would lead to cataclysmic disruptions in markets. Most consumers assume the 18/8 convention.
- Chainlink has a history of going the extra mile to ensure perfect backward compatibility with their upgrades, the now-ancient rounds system is evidence of that.
- I could not find a recommendation in the Chainlink developer documentation to expect this behavior.

Spearbit: Acknowledged.

5.2.4 Single-step `governance` transfer can be risky

Severity: Low Risk

Context: Governable.sol#L39-L41

Description: `Governable.sol` implements the role of `governor` which performs relevant actions like setting oracles for different assets and a possible fallback oracle. It uses a single-step role transfer design, which adds the risk of setting an unwanted role owner by accident if the ownership transfer is not done with excessive care it can be lost forever.

Recommendation: Consider using a two-step ownership transfer mechanism for this critical `governor` change, which would avoid typos, "*fat finger*" mistakes and transfers to the default `address(0)` value.

Additionally, consider using an explicit `renounceOwnership` method to enable the expected renounce method.

Some good implementations of the two-step ownership transfer pattern can be found at Open Zeppelin's `Ownable2Step` or Synthetic's `Owned`.

Euler: Acknowledged. Won't fix.

Spearbit: Acknowledged.

5.3 Gas Optimization

5.3.1 `RedstoneCoreOracle.validateTimestamp` external call block can be improved

Severity: Gas Optimization

Context: RedstoneCoreOracle.sol#L117

Description: The `validateTimestamp` function is `public` and `view` instead of `internal` because it inherits these modifiers from the Redstone core contracts. However, it should only ever be called internally from `updatePrice`. The contract currently sets and checks flags in storage based on if the call comes from `updatePrice`.

Recommendation: Consider removing these flags and checking that the `msg.sig` equals the `RedstoneCoreOracle.updatePrice.selector` in `validateTimestamp` instead. This also ensures that any call to `validateTimestamp` is an internal call that is executed from an `updatePrice` subcall context.

Euler: Fixed in PR 40. The `validateTimestamp` is now a view function.

Spearbit: Verified.

5.4 Informational

5.4.1 Allowing timestamps in the future inside the `RedstoneOracle` could lead to an edge case that should be properly documented

Severity: Informational

Context: RedstoneCoreOracle.sol

Description: The `RedstoneCoreOracle` implementation allows signed price to be at max `1 minute` in the future. When `updatePrice(uint48 timestamp)` is called, the function will early return if the `timestamp` parameter is older compared to the one already approved, validated and cached in `_cache.priceTimestamp`.

Let's assume that at T0 the Redstone signers generate and sign a price with a `timestamp` equal to `T0 + 1min` and that such price is accepted by the `RedstoneCoreOracle` and saved into the `_cache` variable.

This means that for `1 minute`, any price update that is older than `T0 + 1 min` will be discarded. If during this period of time the real price of the asset has oscillations (down/up) compared to the one cached, and all the signed prices have a "normal" timestamp (not in the future), those price updates will be ignored, resulting in possible harm to the protocols that rely on such oracle.

Recommendation: Euler should document this edge case and consider asking the Redstone team for more information regarding the `timestamp-in-the-future` topic to assets which are the side effects of accepting those prices and in which circumstances they should be considerable safe and reliable.

Euler: Acknowledged. According to the RedStone team, a timestamp is allowed to be in the future to accommodate some L2s where `block.timestamp` is taken L1 blocks and not from the L2 sequencer. This issue should not be present on Ethereum given normal functioning of the RedStone network.

Spearbit: Acknowledged.

5.4.2 `RedStone` oracle can be updated with different prices multiple times per transaction

Severity: Informational

Context: RedstoneCoreOracle.sol#L121-L122

Description: The new Redstone code removed the `maxCacheStaleness` code that prevented the oracle from being updated more than once in a transaction. Note that it is currently possible to call `updatePrice` multiple times with different data packages. The only constraint is that the timestamps are in ascending order.

Changing prices multiple times in a single transaction is more risky and could lead to attacks in protocols, for example, in case the oracle is used to mint tokens at one price, and redeem them at a different price for risk-free profit. Note that most oracles, including all TWAP oracles, update at most once per block.

Recommendation: Consider restricting price updates to once per time frame (for example, per block).

Euler: Acknowledged. We settled at not implementing blocking behavior.

Spearbit: Acknowledged.

5.4.3 Liquidation Invariants

Severity: Informational

Context: LiquidityUtils.sol#L73-L120, LTVConfig.sol#L33-L49

Description: Euler uses different LTV configurations as well as different prices for liquidation and borrowing calculations. Borrows are accepted when `healthScore(borrow) > 1.0`, and liquidations are performed when `healthScore(liquidation) <= 1.0` with `healthScore(x) := collateralValue(x) * getLTV(x) / liabilityValue(x)`. It's important that an accepted health check when borrowing does not immediately lead to an unhealthy position regarding liquidation checks. We can prove this by showing `healthScore(liquidation) >= healthScore(borrow)`.

- `getLTV` **invariant:** The following holds for `getLTV`:

  ```
  getLTV(borrowing) <= getLTV(liquidation)
  ```

  Proof: From the code we can distinguish the cases:

  1. `targetLTV >= originalLTV`: `getLTV(borrowing) = getLTV(liquidation) = targetLTV`.
  2. `targetLTV < originalLTV`: `getLTV(borrowing) = targetLTV <= lerp(originalLTV, targetLTV) = getLTV(liquidation)`.

- **Health invariant:** The following holds for `healthScore(x)`:

  ```
  healthScore(borrow) <= healthScore(liquidation)
  ```

Proof:

```
collateralValue(borrow) * getLTV(borrow) / liabilityValue(borrow)
<= collateralValue(liquidation) * getLTV(liquidation) / liabilityValue(liquidation)
```

This follows from:

1. `collateralValue(borrow) <= collateralValue(liquidation)` as borrow uses bid prices compared to liquidations using the mid price.
2. `liabilityValue(borrow) >= liabilityValue(liquidation)` as borrow uses ask prices compared to liquidations using the mid price.
3. `getLTV(borrowing) <= getLTV(liquidation)` by the `getLTV` invariant.

Recommendation: The oracles need to guarantee that the mid-price quote used for liquidations (`oracle.getQuote()`) is indeed in between the (bid, ask) quotes used for the borrows (`oracle.getQuotes()`). The `oracle.getQuotes()` function must also guarantee `bid <= ask`. Document these assumptions on the oracles.

Euler: Acknowledged. We updated the white paper with these assumptions.

Spearbit: Acknowledged.

5.4.4 Missing checks in `CrossAdapter`'s constructor

Severity: Informational

Context: CrossAdapter.sol#L35-L39

Description: The `CrossAdapter` is used for pricing tokens that span across two pairs, for example: One can price `wstETH/USD` by querying a `wstETH/stETH` oracle and a `stETH/USD` oracle, `stETH` being the `cross` token.

The constructor of this contract takes five parameters:

- `base` - the base token.
- `cross` - the token used to bridge the pricing query.
- `quote` - the quote token.
- `oracleBaseCross` - the oracle that resolves base/cross and cross/base.
- `oracleCrossQuote` - The oracle that resolves cross/quote and quote/cross.

These parameters aren't checked properly as follows: the `oracleBaseCross` should support pricing `base/cross` tokens and the `oracleCrossQuote` should support pricing `cross/quote` tokens.

If the following check would be violated, the deployed adapter would be rendered unusable.

Recommendation: Inspecting the adapters we noticed that all of them have a base and a quote address but how to return them can vary, to support multiple ways of returning these addresses, we recommend implementing in the `BaseAdapter` two virtual functions: `getBase/getQuote` which can be overridden in the implementations to return the right addresses.

By using these two new functions, the `CrossAdapter` can then properly check the `base/cross/quote` addresses within the `oracleBaseCross/oracleCrossQuote`.

Furthermore, if these two functions would be implemented, may be worth checking if a `maxStaleness` function should be implemented as well which then can be overridden in the implementations. For the implementations that do not have this requirement, `type(uint256).max` can be returned.

Euler: Acknowledged, won't fix. It is unnecessary to extend the `IPriceOracle` interface to enable construction-time checks in `CrossAdapter`.

Spearbit: Acknowledged.

5.4.5 Inconsistent validation of oracle's returned prices

Severity: Informational

Context: PythOracle.sol#L82, ChainlinkOracle.sol#L52, RedstoneCoreOracle.sol#L73-L74, ChronicleOracle.sol#L55

Description: Euler's various oracles do not consistently validate the prices returned by the oracle. For instance, the `ChainlinkOracle` reverts if the price is less than or equal to zero, whereas the `RedstoneCoreOracle` only ensures that the price does not exceed the maximum allowable value for a `uint208`.

Below is a table summarizing the different validations performed by each oracle:

Oracle	Returned Price Type	Validation
`Chainlink`	`int256`	`price > 0`
`Pyth`	`int64`	`price >= 0`
`Redstone`	`getOracleNumericValueFromTxMsg` returns `uint256`	`price <= type(uint208).max)`
`Chronicle`	`uint256`	`price != 0`
`Uniswap`	N/A	N/A *
`Lido`	N/A	N/A*
`Maker`	N/A	N/A *

Note: The `Uniswap`, `Lido`, and `Maker` oracles do not conduct specialized price validations, as they calculate the prices from on-chain data.

Recommendation: Careful attention should be given to the validation of prices returned from oracles. It is advisable to consider adopting a uniform validation approach across all oracles.

Euler: Fixed in PR 32. Pyth and RedStone now revert if price is 0.

Spearbit: Verified.

5.4.6 The `MIN_TWAP_WINDOW` is set too low

Severity: Informational

Context: UniswapV3Oracle.sol#L20

Description: The `MIN_TWAP_WINDOW` is currently set to 5 minutes, while this is just a minimum boundary check, giving the user the possibility to deploy an oracle with that low TWAP window will render the oracle more risky to TWAP manipulation.

Recommendation: We recommend setting this variable to a minimum of 30 minutes as strongly recommended by various research materials on Uniswap V3:

- Chaos Labs Research
- Uniswap

Euler: Acknowledged.

This analysis assumes that the TWAP oracle will be used for lending markets. However, if we use it for a synths product, a 5-minute TWAP window might be sufficient since the oracle influences a change in funding rates, which does not create immediately realizable value for the attacker.

Spearbit: Acknowledged.

5.4.7 `dai`, `sDai` and `dsrPot` should be hardcoded constant in `SDaiOracle`

Severity: Informational

Context: SDaiOracle.sol#L12-L29

Description: As for Lido, `sDAI` and `dsr` are only available on Ethereum Mainnet, and it could make sense to hard-code all these addresses as `constant`.

Recommendation: Euler should:

- Make both the `dai`, `sDai` and `dsrPot` constant variables and hard-code their addresses. Another option would be to retrieve both the `dai` and `dsrPot` addresses from the `sDai` contract.
- Document that this oracle must be deployed only on the Ethereum Mainnet blockchain.

Euler: Acknowledged. `SDaiOracle` was removed from the codebase, see the issue "`SDaiOracle` could use `sDAI`'s `convert` functions".

Spearbit: Acknowledged.

5.4.8 `stEth` and `wstEth` should be hardcoded constant in `LidoOracle`

Severity: Informational

Context: LidoOracle.sol#L11-L23

Description: While the `wstETH` token exists on L2s as a bridged token, the `stETH` token (Lido contract) does not. The `LidoOracle` can only be deployed on the Ethereum Mainnet, and for such reason, it does not make sense to allow the `constructor` to specify arbitrary values for such contracts.

Recommendation: Euler should:

- Make both the `stEth` and `wstEth` constant variables and hard-code their addresses.
- Document that this oracle must be deployed only on the Ethereum Mainnet blockchain.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.9 Document that the `ChainlinkOracle` `maxStaleness` should be related to the Chainlink price feed heartbeat

Severity: Informational

Context: ChainlinkOracle.sol#L21

Description: The `heartbeat` is a threshold that is used by Chainlink to update the price (triggering a new round) if the price has not changed in heartbeat seconds since the last round. This could be a common scenario in price feeds, where the price is very stable and the Deviation Threshold is big enough.

The `maxStaleness` value should be related to the price feed heartbeat but should be greater than such threshold to allow the Oracle to not revert while Chainlink requests a new round.

Recommendation: Euler should improve the documentation about the `maxStaleness` parameter:

- How the `maxStaleness` should be picked based on the Chainlink price feed `heartbeat`.
- What are the side effects of picking a value that is smaller or greater than the Chainlink price feed `heartbeat`.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.10 Document the implications on the `ChainlinkOracle` for the scenario where Chainlink changes of the heartbeat of a price feed

Severity: Informational

Context: ChainlinkOracle.sol#L54

Description: In `ChainlinkOracle` the `maxStaleness` parameter will be mostly chosen based on the `heartbeat` parameter of a Chainlink price feed. The heartbeat is a threshold that is used to update the price (triggering a new round) if the price has not changed in `heartbeat` seconds since the last round.

Such value is not stored on-chain by Chainlink, but must be retrieved on their documentation or price feed detail web page and could change based on Chainlink will.

If such value changes, the `ChainlinkOracle` contract will suffer from some specific side effects. Let's say that for the pair (`asset1`, `asset2`) the price feed shows a heartbeat of `600` (10 minutes) and Euler decides to set `maxStaleness` to `590` to have some room but still be sure to revert if the price becomes stale.

- Scenario 1) Chainlink decreases the heartbeat from `600` to `300` seconds. This means that after `300` seconds the price will be considered stale by Chainlink and a new round is requested ⇒ As a consequence, Euler's Oracle risks considering "good" a price that in reality should be considered stale because it is indeed stale on Chainlink.
- Scenario 2) Chainlink increases the heartbeat from `600` to `2000` seconds. This means that after `300` seconds, the price will be considered stale by Chainlink and a new round is requested ⇒ As a consequence, Euler's Oracle risks considering stale (and revert) a price that in reality should be considered "good" because it is indeed still valid on Chainlink.

Recommendation: Euler should consider to:

- Make the `maxStaleness` a non-immutable parameter and update it based on Chainlink updates to the price feed heartbeat.
- Document which are the consequences of having an immutable `maxStaleness` when the heartbeat parameter changes on the Chainlink data feed.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.11 Events can be emitted for better monitoring

Severity: Informational

Context: RedstoneCoreOracle.sol#L75-L76

Description: Redstone oracle `updatePrice` changes state variables `cachedPrice` and `cachedTime`.

```
cachedPrice = uint208(price);
cacheUpdatedAt = uint48(block.timestamp);
```

Using an event would improve monitoring for price updates.

Recommendation: Consider adding an event for `updatePrice` to improve monitoring.

Euler: Resolved at file RedstoneCoreOracle.sol in commit 5c5cbf8d.

Spearbit: Verified.

5.4.12 `RedstoneCoreOracle` can only be deployed on the Ethereum mainnet

Severity: Informational

Context: RedstoneCoreOracle.sol#L8-L9

Description: The current implementation of `RedstoneCoreOracle` inherits from `PrimaryProdDataServiceConsumerBase` that exposes two functions:

- `getUniqueSignersThreshold` that represents the number of unique signers who have to sign the price to be accepted.
- `getAuthorisedSignerIndex` that represents the list of signers authorized to sign a price.

Both values will be different depending on which chain the `RedstoneCoreOracle` will be deployed.

Recommendation: Depending on which chain the `RedstoneCoreOracle` will be deployed, it will need to inherit from a different `ConsumerBase` contract of the Redstone SDK.

Euler: Acknowledged. We're currently targeting deployment on Ethereum mainnet, so no fix.

Spearbit: Acknowledged.

5.4.13 `PythOracle` is using `maxStaleness` to validate both price in the past and the future

Severity: Informational

Context: PythOracle.sol#L81

Description: The `_fetchPriceStruct` function of `PythOracle` executes the `PythStructs.Price memory p = IPyth(pyth).getPriceNoOlderThan(feedId, maxStaleness);` call passing `maxStaleness` as an input parameter.

`maxStaleness` is conceived as the maximum number of seconds that the price can be in the past (compared to `block.timestamp`) but in reality, the price could also be in the future.

This fact is corroborated by the logic inside Pyth SDK that performs an abs delta between the `block.timestamp` and the `price.publishTime` in getPriceNoOlderThan. In the **near** SDK the check is even more explicit:

```
pub fn get_price_no_older_than(
    &self,
    price_id: PriceIdentifier,
    age: Seconds,
) -> Option<Price> {
    self.prices.get(&price_id).and_then(|feed| {
        let block_timestamp = env::block_timestamp() / 1_000_000_000;
        let price_timestamp = feed.price.publish_time;

        // - If Price older than STALENESS_THRESHOLD, set status to Unknown.
        // - If Price newer than now by more than STALENESS_THRESHOLD, set status to Unknown.
        // - Any other price around the current time is considered valid.
        if u64::abs_diff(block_timestamp, price_timestamp.try_into().unwrap()) > age {
            return None;
        }

        Some(feed.price)
    })
}
```

Recommendation: Euler should consider using two different state variables to limit a price too much in the past or too much in the future. Instead of using `getPriceNoOlderThan`, `getPriceUnsafe` can be used and the `price.publishTime` can be locally checked against those two boundaries.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.14 `EulerRouter` can be deployed without a governor

Severity: Informational

Context: EulerRouter.sol#L41, Governable.sol#L39-L42

Description: The current implementation of the `Governable` contract does not perform any check on the `_governor` passed in the `constructor`.

Given that `EulerRouter` does not perform any sanity check on the `_governor` inside its `constructor`, it will be possible to deploy an `EulerRouter` that cannot be governed.

Any call to a function that is protected by `onlyGovernor` will revert, and such a router will be useless because it cannot be configurable at all.

Recommendation: Euler should prevent the deployment of an `EulerRouter` with `governor` equal to `address(0)`

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.15 Users can choose best price for `PythOracle`

Severity: Informational

Context: PythOracle.sol#L81

Description: When the `maxStaleness` parameter is large, there will be several prices for a certain `feedId`. Because `Pyth` does not push the data on-chain and lets the users to submit it, one can wait and choose from several prices that are listed by the `Wormhole` and choose the best one that is suited for them.

Attackers might choose the best/worst price for liability/collateral assets and liquidate users.

Another thing to point out is that even if the `maxStaleness` is chosen as right as possible, there is still a possibility of MEV, as pointed out in the `Pyth` documentation as well because in practice the price will be available for MEV to be read off-chain before it is submitted on-chain.

Recommendation: This is a known issue for any Pyth Oracle therefore the `maxStaleness` should be chosen accordingly to avoid this attack vector.

Furthermore, the protocols that wants to use the `PythOracle` can try to avoid the MEV opportunity by implementing the solutions defined by the `Pyth` team (link):

> This analogy suggests two simple solutions to races:
>
> Configure protocol parameters to balance the losses from smart flow against the gains from two-way flow. Market makers in traditional finance implement this approach by offering a bid/ask spread and limited liquidity. The limited liquidity caps the losses to smart flow, while still earning profits from the two-way flow. A successful market maker tunes the spread and offered liquidity to limit adverse selection from smart traders while still interacting with two-way flow.
>
> Give the protocol a "last look" to decide which transactions to accept. In traditional finance, some exchanges give market makers a chance to walk back a trade offer after someone else has requested it. Protocols can implement this technique by splitting transactions into two parts: a request and a fulfillment. In the first transaction, the user requests to perform an action. In the second transaction, the protocol chooses whether or not to fulfill the user's request; this step can be implemented as a permissionless operation. The protocol can require a short delay between the two transactions, and the user's request gets fulfilled at the Pyth price as of the second transaction. This technique gives the protocol extra time to observe price changes, giving it a head start in the latency race.

Euler: Acknowledged. When interacting with pull-based oracles, users control the price update flow. Therefore this issue is a property of both `PythOracle` and `RedstoneCoreOracle`. It is advised to keep `maxStaleness` low.

Spearbit: Acknowledged.

5.4.16 Users can chose best price for `RedstoneCoreOracle`

Severity: Informational

Context: RedstoneCoreOracle.sol#L85-L93

Description: When the `maxPriceStaleness` parameter is large, there will be several data packages (corresponding to a price at a certain time) that are considered valid. As any user can relay the signed data packages to update the oracle, the user can also choose the price that is best suited for them. Attackers might choose the best/worst price for liability/collateral assets and liquidate users.

Recommendation: This is a known issue for any Redstone oracle and the caching recommendation does not mitigate the problem. The `maxPriceStaleness` should be chosen accordingly.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.17 Missing `maxCacheStaleness <= maxPriceStaleness` check

Severity: Informational

Context: RedstoneCoreOracle.sol#L61-L62

Description: When `maxCacheStaleness > maxPriceStaleness`, there will always be a period of at least `maxCacheStaleness - maxPriceStaleness` seconds that the price is considered stale but no new price can be added.

Recommendation: Consider checking `maxCacheStaleness <= maxPriceStaleness` in the constructor as a violation would be a misconfigured oracle.

Euler: Acknowledged. We changed how `RedstoneCoreOracle` is implemented, removing the concept of a cache staleness. Now the adapter records the observation timestamp contained in the price payload itself when updating the price.

Spearbit: Acknowledged.

5.4.18 `SDaiOracle` could use `sDAI`'s `convert` functions

Severity: Informational

Context: SDaiOracle.sol#L36-L55

Description: The `SDaiOracle` implements the same code that `sDAI.convertToAssets/convertToShares` would perform.

Recommendation: Consider directly using `sDAI.convertToAsserts (base == sDai && quote == dai)` and `sDAI.convertToShares (base == dai && quote == sDai)`.

Euler: Acknowledged. We decided to remove `SDaiOracle` from the codebase entirely. Instead, sDai should be configured in EulerRouter as a resolved vault, thus using the ERC4626 pricing in the router. This leaves us without an option to price Dai $\rightarrow$ sDai, however we believe this direction is not necessary.

Spearbit: Acknowledged.

5.4.19 Staleness checks have different meaning for adapters

Severity: Informational

Context: Chainlink, Chronicle, Pyth, Redstone

Description: Several adapters implement staleness checks that compare the current time against the oracle-reported time for that price. However, oracles have different meanings for the returned price timestamps. A staleness check should ideally compare the current time against the time the price was observed (observation timestamp). Some oracles do not provide this information and instead use the timestamp when the price update transaction was mined (mining timestamp) as the timestamp of the price.

Protocol	Returned Timestamp
Chainlink	mining
Chronicle	mining
Pyth	observation
Redstone	observation

Recommendation: Special care must be given to oracles that report the mining timestamp as some transactions might be delayed (network congestion, L2 sequencer downtime, etc.) and the price would already be considered stale otherwise. Consider configuring these adapters with a decreased max staleness limit.

Euler: Partial fix for Chainlink oracle in PR 32.

Spearbit: Only partially fixed.

5.4.20 Fallback oracle is not used when resolved oracle fails

Severity: Informational

Context: EulerRouter.sol#L81

Description: The current semantics of the `fallback` oracle are that it's used as the default oracle if no oracle was registered for the requested pair. It is not used as a fallback in case the resolved oracle reverts.

Recommendation: Consider using the fallback oracle in case the resolved oracle would revert.

Euler: Acknowledged. `fallbackOracle` is a configuration-level fallback as you describe. One useful application of this fallback is configuration overrides. If we consider a "canonical" `EulerRouter` with many well-vetted configurations, one could deploy a custom `EulerRouter` with a few configurations that falls back to the canonical router, essentially inheriting its configurations.

A *backup* oracle is another thing entirely, and a big change in how `EulerRouter` works, so we will not be applying this recommendation.

Spearbit: Acknowledged.

5.4.21 `EulerRouter` vault quotes ignores vault liquidity restrictions

Severity: Informational

Context: EulerRouter.sol#L129

Description: When a `(vault, vault.asset())` pair is configured in `EulerRouter`, it can use the `vault.convertToAssets()` function to get the quote.

Recommendation: Document that the `getQuote*` functions ignore whether the traded amount satisfies the vault's liquidity restrictions, like `vault.maxRedeem() <= inAmount`, slippage/fees that might be part of `previewRedeem()`, or per-user restrictions. The caller is responsible for checking these.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.22 Order of operations in `ScaleUtils.calcOutAmount` can lead to avoidable overflows

Severity: Informational

Context: ScaleUtils.sol#L72

Description: The out amount for the `!inverse` path is computed as `FixedPointMathLib.fullMulDiv(inAmount * unitPrice, priceScale, feedScale)`. The `inAmount` parameter is a user-specified argument and is therefore unbounded. Moving the multiplication from `inAmount * unitPrice` to `priceScale * unitPrice` could lead to fewer overflows in practice as `priceScale` and `unitPrice` are bounded oracle-chosen parameters. It also reflects the symmetry of the computations for both `inverse` conditions better.

Recommendation: Consider changing:

```
  if (inverse) {
      // (inAmount * feedScale) / (priceScale * unitPrice)
      return FixedPointMathLib.fullMulDiv(inAmount, feedScale, priceScale * unitPrice);
  } else {
      // (inAmount * unitPrice * priceScale) / feedScale
-     return FixedPointMathLib.fullMulDiv(inAmount * unitPrice, priceScale, feedScale);
+     return FixedPointMathLib.fullMulDiv(inAmount, priceScale * unitPrice, feedScale);
  }
```

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.23 Oracles and non-crypto price outages

Severity: Informational

Context: IPriceOracle, PythOracle.sol#L81

Description: The `IOracle` interface specifies the use of some special addresses related to ISO 4217 codes. In order to retrieve fiat data, some of these codes would be used under `address(ISO-CODE)`; USD for example would be `address(840)`. A possible issue using oracles arises as some of the price feeds for these codes are retrieved from markets that only work under certain hours. This issue is generic regarding oracles and non-crypto assets and is outlined and related to Pyth's best practices for price availability, Chainlink, Redstone, etc..

Recommendation: Given the nature of price feed outages and their impact, it's advisable to enhance with comments and documentation user awareness, regarding the reliability and potential limitations of price data during unusual market conditions.

Euler: Acknowledged.

Non-crypto assets generally trade during specific market hours, which oracles inherit by nature. While this issue is about Pyth, it's also relevant in Chainlink and Redstone.

Non-crypto assets are incompatible with generalized lending markets not only due to market hours but also because liquidations cannot be carried out on-chain. We consider non-crypto feeds out of scope for the price oracles.

Note: ISO-4217 is supported by `IPriceOracle` primarily to enshrine a specific address for `USD` feeds.

Spearbit: Acknowledged.

5.4.24 `PythOracle#_fetchPriceStruct()` might not work most of the time

Severity: Informational

Context: PythOracle.sol#L81

Description: The feeds on EVM chains for Pyth are updated using the following logic:

1. Publishers are pushing the prices to Pyth Oracle Network (which is deployed on Solana and Pythnet).
2. Wormhole sees an updated price and disperses it to the EVM chains as an off-chain signed message attesting to this.
3. Users have the job to actually publishing this message on-chain so that the `IPyth(pyth).getPriceNoOlderThan(feedId, maxStaleness);` to not revert with Stale Price.

Pyth states clearly that they do not push the updates on EVM chains.

This complicates quite a lot the usage of `PythOracle`, making it very unreliable. Currently, there is no documentation that alerts the user about the unreliability of this oracle.

`PythOracle#_fetchPriceStruct()`'s first line is an external call to `getPriceNoOlderThan`. This call would fail if the price had never been updated:

> This function reverts with a PriceFeedNotFound error if the requested feed id has never received a price update. This error could either mean that the provided price feed id is incorrect, or (more typically) that this is the first attempted use of that feed on-chain. In the second case, calling updatePriceFeeds will solve this problem.

Or fail in the cases where this last update it's too old, meaning a brick in all services unless externally we call some of the `updatePriceFeeds` functions:

> The caller provides an age argument that specifies how old the price can be. The call reverts with a StalePriceError if the on-chain price is from more than age seconds in the past (with respect to the current on-chain timestamp). Call updatePriceFeeds to pull a fresh price on-chain and solve this problem.

Recommendation: Our recommendation is to add a warning just as `UniswapV3Oracle` has related to the fact that if one wants to integrate with it, one has the following options:

- Get the off-chain signature from Wormhole and publish it on-chain for the oracle to be up to date using `IPyth.updatePriceFeeds*`.
- Wait until someone else does that, this option being the most risky one as the waiting time can vary a lot.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.25 Sequencer availability is not validated

Severity: Informational

Context: ChainlinkOracle.sol#L48

Description: For L2s, `_getQuote` is not following the standard Chainlink approach. The issue here is related to be able to include transactions with the sequencer down via `forceInclusion`, allowing the possibility to use outdated oracle versions and their prices to, for example, borrow more assets than what should be possible. Even if Chainlink Oracle price feed updates should be done, until the sequencer availability is restored, it won't be able to update correctly the prices on L2s with sequencers down, allowing the messages sent by the `delayedInbox` to be profitable.

The expected design is not only to validate `dataFeed` timestamps in order to avoid stale data but to add a `sequencerFeed` address which should be validated too in order to verify that the sequencer is up, and therefore different oracle versions would not be used.

Recommendation: To make Chainlink oracle adapter code compatible with L2s such as Arbitrum, sequencer checks should be included for the L2 contract

Euler: Acknowledged. Won't be adding a sequencer liveness oracle yet because we're currently targeting deployment on Ethereum mainnet.

Spearbit: Acknowledged.

5.4.26 "*Magic numbers*" should be defined as constants to improve readability and maintainability

Severity: Informational

Context: PythOracle.sol#L82

Description: Numbers not defined as constants are less maintainable and readable. Variables such as `10000` for basis points can be defined as `BASIS_POINTS` to ease the read and search.

Recommendation: Use named constants for magic numbers over the code in order to keep a cleaner and more maintainable codebase.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.27 `PythOracle` should bound `_maxConfWidth` value to a min/max value

Severity: Informational

Context: PythOracle.sol#L54

Description: `_maxConfWidth` is a basis points value which is upper bounded to `10000` (which represents 100%). However, the current implementation of the `PythOracle constructor` does not impose any lower or upper limit to such value.

- Unless `p.conf` (price confidence) is equal to `0` (high unlikely), having `_maxConfWidth = 0` would mean that any call to the oracle will revert because of `PriceOracle_InvalidAnswer`.
- Allowing `_maxConfWidth` to be `>= 10_000` (100%) would mean allowing the price to diverge more than the price itself.

Furthermore, the manner in which the confidence is chosen should be better documented for the `PythOracle` for the users to make the best value:

- For pairs that are highly correlated (such as UDSC/USD) one should use a smaller maxConfWidth.
- For pairs that are more volatile but represent currencies that are well known (such as BTC/USD) a more permissive `maxConfWidth` should be chosen.
- For highly volatile currencies, like new currencies, a very permissive `maxConfWidth` should be chosen.

Recommendation: Euler should consider adding safe lower and upper bound limits to the value that `maxConfWidth` should assume to prevent unexpected reverts or price accepted with a too high confidence.

Consider adding extra documentation about how to choose the right value for this value. Furthermore, the Euler team can perform some simulations or provide some researched values and present them to the users to best select this value.

Euler: Fixed in PR 32.

Spearbit: Verified.

5.4.28 Missing safety checks can lead to undesired behaviors

Severity: Informational

Context: ChainlinkOracle.sol#L30, ChronicleOracle.sol#L33, LidoOracle.sol#L20, SDaiOracle.sol#L25, PythOracle.sol#L41-L48, RedstoneCoreOracle.sol#L49-L66, UniswapV3Oracle.sol#L39, CrossAdapter.sol#L34, Governable.sol#L19-L27, EulerRouter.sol#L41-L48

Description: In different contract constructors and some setters, different addresses are set to a variable without checking whether these addresses are non-zero, or if they represent deployed contracts. This contrasts with the approach in other contracts where such safety checks are implemented. Ignoring these checks could lead to undesired behavior.

- `ChainlinkOracle constructor` sanity checks:
  - `_base !== address(0)`.
  - `_quote !== address(0)`.
  - `_base !== _quote`.
  - `_feed !== address(0)`.
  - Add an upper bound limit to the `_maxStaleness` that should be greater than the max `heartbeat` used by Chainlink in their feed. Such bound should consider possible changes to the `heartbeat` parameter.
  - Add a lower bound to `_maxStaleness`, values like `_maxStaleness == 0` would only allow the oracle to work with prices that have been updated in the very same tx of call to `_quote`.
- `ChronicleOracle constructor` sanity checks:
  - `_base !== address(0)`.
  - `_quote !== address(0)`.
  - `_base !== _quote`.
  - `_feed !== address(0)`.
  - Add an upper bound limit to the `_maxStaleness`.
  - Add a lower bound to `_maxStaleness`, values like `_maxStaleness == 0` would only allow the oracle to work with prices that have been updated in the very same transaction calling `_quote`.
- `PythOracle constructor` sanity checks:
  - `_base !== address(0)`.
  - `_quote !== address(0)`.
  - `_base !== _quote`.
  - `_pyth !== address(0)`.
  - Add an upper bound limit to the `_maxStaleness`.
  - Add a lower bound to `_maxStaleness`, values like `_maxStaleness == 0` would only allow the oracle to work with prices that have been updated in the very same transaction calling `_quote`.

- Add an upper bound limit to the `_maxConfWidth` to be less than `10_000`.
- Add a valid lower bound limit to the `_maxConfWidth`. With `_maxConfWidth = 0`, every `_fetchPriceStruct` execution will revert (unless `p.conf = 0`, which is very unlikely).

• `RedstoneCoreOracle constructor` sanity checks:
  - `_base !== address(0)`.
  - `_quote !== address(0)`.
  - `_base !== _quote`.
  - Add an upper bound limit to the `_maxPriceStaleness`.
  - Add a lower bound to `_maxStaleness`, values like `_maxStaleness == 0` would only allow the oracle to work with prices that have been updated in the very same transaction calling `_quote`.
  - Add an upper bound limit to the `_maxCacheStaleness`.

• **UniswapV3Oracle** `constructor` sanity checks:
  - `_tokenA !== address(0)`.
  - `_tokenB !== address(0)`.
  - `_tokenA !== _tokenB`.

• `EulerRouter constructor` sanity checks:
  - `_governor !== address(0)` to prevent the deployment of a non-configurable Oracle router.

Recommendation: Consider adding the checks suggested above.

Euler: Fixed the following cases in PR 32:

• Upper and lower bounds for `maxStaleness` in `ChainlinkOracle` and `ChronicleOracle`.
• Upper bound for `maxStaleness` in `PythOracle`.
• Upper and lower bounds for `maxConfWidth` in `PythOracle`.
• Upper bound for `maxPriceStaleness` in `RedstoneOracle`.
• Upper bound for `maxCacheStaleness` in `RedstoneOracle`.

The rest are acknowledged.

Spearbit: Verified.

5.4.29 Missing/wrong comments and typos affect readability

Severity: Informational

Context: See each case below

Description: Comments help to provide context and documentation on what different functions, contracts and variables do. Providing clear and precise comments is key to a clean and maintainable codebase. See below a list of related nitpicks:

• Unclear comments:
  - PythOracle.sol#L78-L79: The line is unnecessary cut at the middle of a comment.
  - CrossAdapter.sol#L43-L44: Two `@dev` explanations are switched in order to explain an `if-else` block functioning.

• Wrong comments:
  - IPriceOracle.sol#L20: `The amount of "quote" you would get for buying...` should be `The amount of "quote" you would spend for buying....`

    - ChronicleOracle.sol#23: `Reverts if age > maxStaleness` should be `Reverts if block.timestamp - age > maxStaleness`.
    - RedstoneCoreOracle.sol#L81: "/// @dev This function will be called in `getOracleNumericValueFromTxMsg` in `getQuote`", should be "/// @dev This function will be called in `getOracleNumericValueFromTxMsg` in `updatePrice`".
    - PythOracle.sol#L70: `// priceStruct.expo will always be negative` should be `// priceStruct.expo will always be non-positive` as `0` is indeed valid.
- Typos:
    - UniswapV3Oracle.sol#17: `accomodate` should be `accommodate`.
- Natspec `@return` missing:
    - PythOracle.sol#L80 for unnamed return struct .
- Natspec `@param` missing
    - ScaleUtils.sol#L22: `priceExponent`, `feedExponent` are missing.
- Natspec missing: IChronicle.sol#L5, IChronicle.sol#L6, IStEth.sol#L5, IStEth.sol#L6, BaseAdapter.sol#L34, AggregatorV3Interface.sol#L5, AggregatorV3Interface.sol#L6, AggregatorV3Interface.sol#L7, IPot.sol#L5, IPot.sol#L6, IPot.sol#L7, ScaleUtils.sol#L13, SDaiOracle.sol#L12

Recommendation: Improve comments all over the code, correct typos, remove stale comments and fix incorrect comments where possible.

Euler: Resolved in PR 32.

Spearbit: Verified.