Zellic

Prepared for
Tyler Loewen
Adrastia

Prepared by
Nipun Gupta
Vlad Toie
Zellic

January 12, 2024

Adrastia Protocol

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana, as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional infosec and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for Adrastia from December 11th, 2023 to January 12th, 2024. During this engagement, Zellic reviewed Adrastia Protocol's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How does Adrastia collect and store the data for the price feeds?
- Is access control properly implemented? Who can update the price feeds?
- How are the components of the protocol structured? What are the dependencies?
- Are there any issues related to data freshness in the price feeds? How is this handled?
- Are there proper guidelines for the usage of the protocol's functions?

1.2. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Any off-chain components that perform validation checks on price or liquidity data
- Management of the privileged accounts that interact with the protocol

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, the limited amount of time for the assessment meant that we could not formally verify the correctness of all the arithmetic operations in the codebase.

Moreover, due to the centralization of the off-chain components tasked with prior validation of data, it is not possible to comprehensively assess the entire security posture of the project with complete certainty. For example, if the off-chain components are compromised, the protocols that are using the price feeds will be drastically affected.

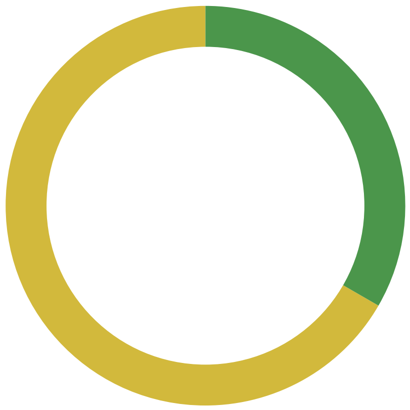
1.3. Results

During our assessment on the scoped Adrastia Protocol contracts, we discovered six findings. No critical issues were found. Four findings were of medium impact and two were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Adrastia's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	4
🟩 Low	2
⬜ Informational	0



2. Introduction

2.1. About Adrastia Protocol

Adrastia filters, validates, and aggregates on-chain data to provide robust data feeds.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Adrastia Protocol Contracts

Repositories	https://github.com/adrastia-oracle/adrastia-core ↗ https://github.com/adrastia-oracle/adrastia-periphery ↗
Versions	adrastia-core: `0e7b1a7222dc835cd5f1cfaac8f4ec6f33a8b0ea` adrastia-periphery: `d7cdc872549c76410f21fe9666f83f8ea8078610`
Programs	• adrastia-core/contracts/* • excluding: test/*,libraries/balancer-v2/*, libraries/uniswap-lib/* • adrastia-periphery/contracts/* • excluding: test/*, vendor/*
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 4, 2023	Kick-off call
December 11, 2023	Start of primary review period
January 12, 2024	End of primary review period

3. Detailed Findings

3.1. Consult should fail when update fails

Target	Project Wide		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

Across multiple contracts, the `update` function is used to store new observations of prices or liquidity into the system. This function is generally called by privileged entities and involves querying a specific oracle / data source for the new observation rather than manually inputting it.

We note a general concern here, in that even though the `update` function might fail and revert, the `consult` function, which queries data that the `update` function is supposed to update, might still succeed. This means that even though the data has not been updated as intended, the `consult` function will still return the old data, which might be stale and therefore not representative of the current market conditions.

Currently, the partial-implemented mitigation against stale data is to check the `maxAge` of the observation before using it — for example, the `consultLiquidity` function in `LiquidityAccumulator`:

```
function consultLiquidity(
    address token,
    uint256 maxAge
) public view virtual override returns (uint112 tokenLiquidity,
    uint112 quoteTokenLiquidity) {
    // ...

    require(observation.timestamp != 0, "LiquidityAccumulator:
    MISSING_OBSERVATION");
    require(block.timestamp <= observation.timestamp + maxAge,
    "LiquidityAccumulator: RATE_TOO_OLD");

    // ...
}
```

This means that the `consult` function will only return the observation if it is not stale, time-wise. However, it's important to note that this alone is not a comprehensive mitigation, as the data may still be stale in terms of relevance. Even if the timestamp of the previously stored observation is relatively recent, the information might significantly deviate from the current market value, rendering it non-representative of the present market conditions.

To give a more concrete example, here we have the implementation of `performUpdate` in `PeriodicPriceAccumulationOracle`:

```
function performUpdate(bytes memory data)
    internal virtual override returns (bool) {

    address token = abi.decode(data, (address));
    uint256 gracePeriod = accumulatorUpdateDelayTolerance();

    if (
        IUpdateable(priceAccumulator).timeSinceLastUpdate(data) >=
        IAccumulator(priceAccumulator).heartbeat() + gracePeriod
    ) {
        revert PriceAccumulatorNeedsUpdate(token);
    }

    AccumulationLibrary.PriceAccumulator memory priceAccumulation
    = IPriceAccumulator(priceAccumulator)
        .getCurrentAccumulation(token);

    return priceAccumulation.timestamp != 0 && push(token,
    priceAccumulation);
}
```

This will revert if the `priceAccumulator` has not been updated within the grace period. However, the `consultPrice` / `consultLiquidity` functions from `PeriodicPriceAccumulationOracle` will still return the old observation, even though the `performUpdate` with the intended new observation has failed.

Impact

The data that is consulted might be stale and therefore drastically different than its actual real values. This could lead to the protocol providing inaccurate price and liquidity data, which could in turn lead to other protocols that rely on this data to make incorrect decisions.

Recommendations

We recommend that once the `update` function fails, all the `consult` functions should be paused, so that the data that is being consulted is always fresh and therefore representative of the current market conditions. Moreover, we recommend that whenever the `update` function is paused, the `consult` functions should also be paused.

We understand that this might be a deliberate design choice; however, we believe that considering a more conservative approach, where the data is not consulted if it is not fresh, would be more beneficial for the protocol and its users.

Remediation

This issue has been acknowledged by Adrastia. The team has also provided the following clarification over this issue:

> At the aggregator level with respect to underlying `PeriodicAccumulationOracle` and `PeriodicPriceAccumulationOracle` instances, if the update fails, the later call to `consult(token, maxAge)` will subsequently fail and the underlying oracle's observation will be discarded.
>
> Furthermore, data availability is a priority, and having `consult` fail when `update` fails could open up additional attack vectors.
>
> Finally, it should be noted that these contracts compute time-weighted averages. TWAs describe the past and may not be reflective of the present.

3.2. Precision loss in rate calculation

Target	RateController		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `computeRateInternal` calculates the rate for the specified token by summing its base rate and the weighted rates of its components. The weighted rates are divided by 10,000 (100%) first to convert them into percentages and then added to the `componentRate` variable, which is then added to the final rate. In the code, 100% is represented as 10,000, and dividing componentWeights by 10,000 ensures that the weighted rates are expressed as a percentage of the total.

```
function computeRateInternal(address token)
    internal view virtual returns (uint64) {
        RateConfig memory config = rateConfigs[token];

        uint64 rate = config.base;

        for (uint256 i = 0; i < config.componentWeights.length; ++i) {
            uint64 componentRate
    = ((uint256(config.components[i].computeRate(token))
    * config.componentWeights[i]) /
                10000).toUint64();

            rate += componentRate;
        }

        return rate;
    }
```

If the value is not adequately scaled within the `computeRate` function, the resulting rate may be lower than expected. The issue might be amplified in case of high-value tokens like BTC. The `computeRate` function utilizes the value returned by `getValue` to compute the clamped rate, and `getValue` divides the value by the token decimals before returning it. Consequently, if the scaling is insufficient, the `computeRate` function might return the rate in the same decimal notation. Even a minor precision loss could result in a substantial difference in the rate for a high-value token, given that a single token may be valued in the thousands in USDC

Consider the following example: assume ten components, each with component weight of `1000` and the value returned from `computeRate` for each of these components is `9`. According to the rate calculation mentioned above, the value of `componentRate` for each of these components would be calculated as follows:

```
uint64 componentRate = ((9 * 1000) / 10000).toUint64();
```

Due to the precision loss, the value of `componentRate` for each of these components will be `0` and the final rate will be `config.base + 0`. Conversely, if the component rates were added first and then divided by 10,000, the final rate would be `config.base + 8`. In this example, if the token used is BTC, and the value returned from `computeRate` is without token decimals, the rate returned by `computeRateInternal` is 8 BTC less than what is expected, resulting in a significant discrepancy.

Impact

The value of rate returned from the above function might be less than the expected value.

Recommendations

We recommend adding the weighted rates first and then dividing by 10,000. Furthermore, we advise carefully configuring the parameters for each component to ensure that the returned value from `computeRate` is adequately scaled, minimizing any potential precision loss.

Remediation

This issue has been acknowledged by Adrastia, and a fix was implemented in commit 86f7eded ↗.

3.3. Update threshold is not dynamic

Target	AbstractAccumulator		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The liquidity and price accumulators are responsible for storing price and liquidity data for the protocol. This data is stored as an observation, which is regularly updated by privileged entities through the use of the `update` function. Whenever an `update` call occurs, the new observation is validated against specific constraints, such as time since last update, threshold between new and old observation, and so on. If the new observation is valid, it is stored as the current observation.

It is important to note here that the `updateThreshold`, or the minimum difference between the new and old observation, is not dynamic. This means that the update threshold is fixed at the time of deployment, and cannot be changed afterwards. This becomes problematic under high volatility market conditions, as the update threshold might be too high to allow for updates to occur, which would in turn result in the contract being unable to update its observations.

Impact

Due to the fixed update threshold, the contracts might not be adequately updatable during high volatility market conditions, therefore drastically limiting the usability of the contracts. Observations will not update properly, and the protocol would therefore not be able to provide accurate price and liquidity data.

Recommendations

We recommend implementing updating the AbstractAccumulator so that `theUpdateThreshold` is no longer immutable. Additionally, we recommend implementing a function that allows the owner to update the update threshold, so that it can be changed dynamically, depending on market conditions.

```
abstract contract AbstractAccumulator is IERC165, IAccumulator {
    uint256 internal immutable theUpdateThreshold;
    uint256 internal theUpdateThreshold;

    // ...
    function setUpdateThreshold(uint256 newUpdateThreshold) external
        virtual onlyOwner {
        theUpdateThreshold = newUpdateThreshold;
    }
     // ...
}
```

Note that this issue is mitigated in the `Managed` versions of the accumulators, as the owner can update the threshold through the `setConfig` function, located in the `AccumulatorConfig` contract. Therefore, assuming that all the Accumulators will eventually be deployed as `Managed` versions, this issue will not be present in the final product.

Remediation

This issue has been acknowledged by Adrastia. The team has also provided the following clarification over this issue:

> This functionality, introduced in v4, is provided by the Adrastia Periphery contracts by overriding `_updateThreshold()`. Those with the CONFIG_ADMIN role can update this threshold. These managed contracts are used in all current and past deployments.

3.4. Uninitialized Uniswap V3 pool could be used for price manipulation

Target	UniswapV3PriceAccumulator		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The contract UniswapV3PriceAccumulator accumulates prices from the Uniswap pool of tokens and quote tokens with different fees and uses harmonic mean weighted by in-range liquidity to calculate the final price. The function `calculateWeightedPrice` iterates though the different fee values and calculates the weighted harmonic mean. If the pool of a certain fee value is not initialized, the function will skip that pool and go over the pool with the next fee value.

```
function calculateWeightedPrice(address token)
    internal view virtual returns (bool hasLiquidity, uint256 price) {
        uint24[] memory _poolFees = poolFees;
        uint128 wholeTokenAmount = computeWholeUnitAmount(token);
        uint256 numerator;
        uint256 denominator;

        for (uint256 i = 0; i < _poolFees.length; ++i) {
            address pool = computeAddress(uniswapFactory, initCodeHash,
    getPoolKey(token, quoteToken, _poolFees[i]));

            if (pool.isContract()) {
                uint256 liquidity = IUniswapV3Pool(pool).liquidity(); //
    Note: returns uint128
                if (liquidity == 0) {
                    continue;
                }
                liquidity = liquidity << 120;
                (uint160 sqrtPriceX96, , , , , , )
    = IUniswapV3Pool(pool).slot0();
                uint256 poolPrice = calculatePriceFromSqrtPrice(token,
    quoteToken, sqrtPriceX96, wholeTokenAmount) + 1;

                numerator += liquidity;
                denominator += liquidity / poolPrice;
            }
        }
        if (denominator == 0) {
            return (false, 0);
        }
        return (true, numerator / denominator);
    }
```

However, it is possible for anyone to initialize a Uniswap pool of certain fee value if it is not already initialized. Furthermore, they can set the first `sqrtPriceX96` value too during the initialization. An attacker can initialize a pool with an incorrect `sqrtPriceX96` value and a small amount of liquidity such that it cannot be profitable for MEV bots but large enough that it can be used to manipulate the price returned by `calculateWeightedPrice`.

Impact

An attacker could manipulate the price returned by UniswapV3PriceAccumulator. Although the price manipulation will be capped by the threshold of that oracle, it could still lead to issues for protocols that integrate this oracle. Specifically for protocols where small price changes could lead to significant impact, for example lending protocols or trading platforms.

Recommendations

We recommend to set the `poolFees` only after analyzing those pools and verifying that they have enough liquidity such that the price could not be manipulated.

Remediation

This issue has been acknowledged by Adrastia. The team has provided the following clarification over this issue:

> Across Arbitrum, Optimism, and Polygon where Uniswap v3 pools are aggregated, all pools are initialized. The least liquid pool has about $421k in TVL, namely the USDC/WETH 0.3% pool on Polygon.
>
> From now on, new feeds and deployments will utilize one and only one fee tier to prevent low liquidity pools from compromising the data of the other Uniswap v3 pools. Pools are carefully considered based on the security considerations of the DEX protocol, TVL, trading volume, and LP distribution.

3.5. The function `setConfig` may revert under valid circumstances

Target	RateController		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The comment in the function `setConfig` suggests that it is possible to have a sum of component weights equal to 100% and base rate as a nonzero value. However, the function would revert in such scenario.

```
function setConfig(address token, RateConfig calldata config)
    external virtual {
    //...
    // Ensure that the sum of the component weights less than or equal to
    10000 (100%)
    // Notice: It's possible to have the sum of the component weights be
    less than 10000 (100%). It's also possible
    // to have the component weights be 100% and the base rate be non-zero.
    This is intentional because we don't
    // have a hard cap on the rate.

    //...

    // Ensure that the base rate plus the sum of the maximum component rates
    won't overflow
    if (uint256(config.base) + ((sum * type(uint64).max) / 10000) >
    type(uint64).max) {
        revert InvalidConfig(token);
    }

    //...
}
```

In the above inequality, if the value of the sum is 10,000 (100%) and `config.base` is greater than zero, the value of the left hand side would be greater than `type(uint64).max` and the function would revert.

Impact

The function `setConfig` would revert if the config has components as 100% and `config.base` in a nonzero value.

Recommendations

As the value of `computeRate` is clamped to `config.max` in the `computeRate`, it is not possible for `computeRate` to return `type(uint64).max` until `config.max` is set to the same value. We suggest using the `configs[token].max` value from MutatedValueComputer instead of `type(uint64).max` to check in overflow.

Remediation

This issue has been acknowledged by Adrastia, and a fix was implemented in commit 9151d87b ↗.

The team has also commented on the issue:

> `MutatedValueComputer` is not the only rate computer that may be used with `RateController`, so we cannot reference functions outside of the `IRateComputer` interface. Other implementations of `IRateComputer` are not included in this repository and are out-of-scope.
>
> The comment in `setConfig` is wrong and was written before the revert was added. After some testing, it was discovered that if the config has components as 100% or more and config.base in a nonzero value, `update` could revert when it should clamp to the max value. The revert in `setConfig` was then added to prevent this. The outdated comment has been removed.

3.6. Repeated entries in `config.components` might lead to incorrect rate

Target	RateController		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `setConfig` in RateController is used to set/update the rate configuration of a token. The `RateConfig` struct consists of various components and is used to compute rate by calling `computeRate` on these components. The `setConfig` function does not check for repeated entries for these components.

```
struct RateConfig {
        uint64 max;
        uint64 min;
        uint64 maxIncrease;
        uint64 maxDecrease;
        uint32 maxPercentIncrease; // 10000 = 100%
        uint16 maxPercentDecrease; // 10000 = 100%
        uint64 base;
        uint16[] componentWeights; // 10000 = 100%
        IRateComputer[] components;
    }

function setConfig(address token, RateConfig calldata config)
    external virtual {
        checkSetConfig();

        //...
        uint256 sum = 0;
        for (uint256 i = 0; i < config.componentWeights.length; ++i) {
            if (
                address(config.components[i]) == address(0) ||

    !ERC165Checker.supportsInterface(address(config.components[i]),
    type(IRateComputer).interfaceId)
            ) {
                revert InvalidConfig(token);
            }

            sum += config.componentWeights[i];
        }
        if (sum > 10000) {
            revert InvalidConfig(token);
        }

        //...
        rateConfigs[token] = config;
        //...
    }
```

Impact

If the config consists of repeated entries of a component, it will lead to an incorrect rate calculation in the function `computeRateInternal`.

```
function computeRateInternal(address token)
    internal view virtual returns (uint64) {
    RateConfig memory config = rateConfigs[token];

    uint64 rate = config.base;

    for (uint256 i = 0; i < config.componentWeights.length; ++i) {
        uint64 componentRate
    = ((uint256(config.components[i].computeRate(token))
    * config.componentWeights[i]) /
            10000).toUint64();
        rate += componentRate;
    }

    return rate;
}
```

Recommendations

We recommend adding a check that the components do not have repeated entries.

Remediation

This issue has been acknowledged by Adrastia, and a fix was implemented in commit 4d10dd80 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Cumulative prices might be inaccurate if the bot does not call `update` often

The PriceAccumulator accumulates prices across time. It does that mainly by storing the previous observation's price and multiplying it by the delta in time between the previous and current observation timestamps. This is done in the `performUpdate` function, within the PriceAccumulator. In the case of ArithmeticAveraging, the formula for this time-weighted price is as follows,

```
value * weight
```

where `value` is the previous observation's price and `weight` is the delta in time between the previous observation and the current one's timestamp.

The potential problem here is that the `performUpdate` function is only called when the `update` function is called. This means that if the `update` function is not called often enough, the delta in time between the previous observation and the current one's timestamp might be too large, and therefore the time-weighted price might be inaccurate, as the actual price spread across the delta in time might not actually be linear. This problem is mitigated as long as the prices are updated often enough. This potential issue is replicable in the other accumulators as well, such as in the LiquidityAccumulator.

4.2. Prices on Arbitrum could not be updated every block

The function `needsUpdate` checks if the `deltaTime` (or the time since last update) is less than `_updateDelay()` and thus ensures updates occur at most once every `minUpdateDelay`. The value of `minUpdateDelay` is 1 as per the contracts deployed on chain.

As the block time of Arbitrum is ~0.25 seconds, the update could only be called at most once every 3–4 blocks as compared to prices on other chains that could be updated every block. This could be an issue for integrators who want more frequent price updates (e.g., leverage trading platforms).

4.3. The comment in RateController needs to be updated

The comment in RateController suggests to add access control to these four functions: `checkSetConfig`, `checkSetUpdatesPaused`, `checkSetRatesCapacity`, and `checkUpdate`. However, it is missing an important function, `checkManuallyPushRate`.

```
// @dev This contract is abstract because it lacks restrictions on sensitive
    functions. Please override checkSetConfig,
// checkSetUpdatesPaused, checkSetRatesCapacity, and checkUpdate to add
    restrictions.
```

We recommend updating the comment to add the function `checkManuallyPushRate` in the list of functions to which restrictions should be added.

In our assessment of Adrastia's test suite, we observed that while it provides adequate coverage for many aspects of the codebase, there are specific branches and code paths that appear to be under-tested or not covered at all.

Function `setConfig`'s Edge Cases: Function `setConfig`, a critical component of the system used for defining the configuration of the RateControllers and other components of the protocol, seems to be well-tested for its typical use cases. However, we've noticed a gap in testing when it comes to edge cases. These include scenarios like assuring that each individual `componentWeight` is not 0, or even assuring that the sum of `componentWeights` is not 0. Similarly, we have noted uncovered scenarios in the threat model section, which we believe should be tested. It's essential to include tests for these scenarios to ensure the function behaves predictably under all conditions.

Smart Contract State Transitions: The test suite effectively verifies common state transitions in the smart contract. However, we noted that certain uncommon or exceptional state transitions aren't adequately tested. Ensuring the correctness of these transitions is essential to maintain the integrity of the system. We therefore recommend improving test coverage around the freshness of the observations, the validity of the observations, and the states in which the protocol is in for various uncommon scenarios, such as high or low grace periods, high or low volatility(for cummulative weighted prices), and so on.

In our assessment, we found that enhancing test coverage for these specific areas would further bolster the reliability and resilience of the project. We recommend expanding the test suite to include test cases addressing the above points.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

Please note that our threat model was based on commits 0e7b1a72 ↗ and d7cdc872 ↗, which represent a specific snapshot of the codebase. Therefore, it's important to understand that the absence of certain tests in our report may not reflect the current state of the test suite.

During the remediation phase, Adrastia took proactive steps to address the findings by adding test cases where applicable. They also ensured that previously uncovered code branches were thoroughly tested with multiple tests. This demonstrates their dedication to enhancing the code quality and overall reliability of the system, which is commendable.

5.1. Module: AaveCapController.sol

Function: `pushToConfigEngine(address token, RateLibrary.Rate rate)`

This allows the rate controller to push the rate to the Aave Config Engine.

Inputs

- `token`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None. Assumed that the token is registered in the Aave Config Engine.
  - **Impact**: The token whose config is being updated.
- `rate`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None. Assumed to be a valid rate.
  - **Impact**: The rate to push to the Aave Config Engine.

Branches and code coverage

Intended branches

- Call `IAaveV3ConfigEngine.updateCaps()` with the configured parameters.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the rate controller. This should be enforced in the functions that call this function (e.g., RateController's `manuallyPushRate`).
  - ☑ Negative test

5.2. Module: AaveOracleMutationComputer.sol

Function: `checkSetConfig()`

A function that checks whether the `msg.sender` has the required role to set the config.

Branches and code coverage

Intended branches

- Revert if `msg.sender` does not have the `POOL_ADMIN` role.
  - ☑ Test coverage
- Assume it is called in every `setConfig` function from underlying contracts.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the pool admin to set the config.
  - ☑ Negative test

5.3. Module: AaveV3BorrowMutationComputer.sol

Function: `checkSetConfig()`

A function that checks whether the `msg.sender` has the required role to set the config.

Branches and code coverage

Intended branches

- Revert if `msg.sender` does not have the `POOL_ADMIN` role.
  - ☑ Test coverage
- Assume it is called in every `setConfig` function from underlying contracts.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the pool admin to set the config.
  - ☑ Negative test

5.4. Module: AaveV3SupplyMutationComputer.sol

Function: `checkSetConfig()`

A function that checks whether the `msg.sender` has the required role to set the config.

Branches and code coverage

Intended branches

- Revert if `msg.sender` does not have the `POOL_ADMIN` role.
  - ☑ Test coverage
- Assume it is called in every `setConfig` function from underlying contracts.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the pool admin to set the config.
  - ☑ Negative test

5.5. Module: AbstractOracle.sol

Function: `consultLiquidity(address token)`

This returns the token and quote-token liquidity from the latest observation.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The token for which liquidity is returned.

Branches and code coverage

Intended branches

- Return `0,0` if `token` is a quote token.
  - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test

Function: `consultLiquidity(address token, uint256 maxAge)`

This returns the liquidity from the latest observation if `maxAge` is greater than zero. The maximum age of observation is bound to `maxAge`. If `maxAge` is zero, the function calls `instantFetch` to return the real-time liquidity.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The token for which liquidity is returned.
- `maxAge`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The maximum age of the quotation, in seconds. If zero, fetches the real-time liquidity.

Branches and code coverage

Intended branches

- Return `0,0` if `token` is a quote token.
    - ☑ Test coverage
- Call `instantFetch` to fetch the real-time liquidity if `maxAge` is zero.
    - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
    - ☑ Negative test
- Revert if `observation.timestamp + maxAge` is less than `block.timestamp`.
    - ☑ Negative test

Function: `consultPrice(address token)`

This returns the price from the latest observation.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.

        - **Impact**: The token for which the price is returned.

Branches and code coverage

Intended branches

- Return `10 ** quoteTokenDecimals()` if `token` is a quote token.
    - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
    - ☑ Negative test

Function: `consultPrice(address token, uint256 maxAge)`

This returns the price from the latest observation if `maxAge` is greater than zero. The maximum age of observation is bound to `maxAge`. If `maxAge` is zero, the function calls `instantFetch` to return the real-time price.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The token for which the price is returned.
- `maxAge`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The maximum age of the quotation, in seconds. If zero, fetches the real-time price.

Branches and code coverage

Intended branches

- Return `10 ** quoteTokenDecimals()` if `token` is a quote token.
    - ☑ Test coverage
- Call `instantFetch` to fetch the real-time price if `maxAge` is zero.
    - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
    - ☑ Negative test

- Revert if `observation.timestamp + maxAge` is less than `block.timestamp`.
  - ☑ Negative test

Function: `consult(address token)`

This returns the price, token liquidity, and quote-token liquidity from the latest observation.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The token for which price and liquidity is returned.

Branches and code coverage

Intended branches

- Return `10 ** quoteTokenDecimals(),0,0` if `token` is a quote token.
  - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test

Function: `consult(address token, uint256 maxAge)`

This returns the price and liquidity from the lastest observation if `maxAge` is greater than zero. The maximum age of observation is bound to `maxAge`. If `maxAge` is zero, the function calls `instant-Fetch` to return the real-time price and liquidity.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The token for which price and liquidity is returned.
- `maxAge`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The maximum age of the quotation, in seconds. If zero, fetches the real-time price and liquidity.

Branches and code coverage

Intended branches

- Return `10 ** quoteTokenDecimals(),0,0` if `token` is quote token.
  - ☑ Test coverage
- Call `instantFetch` to fetch the real-time price and liquidity if `maxAge` is zero.
  - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test
- Revert if `observation.timestamp + maxAge` is less than `block.timestamp`.
  - ☑ Negative test

5.6. Module: AccumulatorConfig.sol

Function: `initializeRoles()`

This initializes the roles for the contract.

Branches and code coverage

Intended branches

- Should set the deployer as the admin.
  - ☑ Test coverage
- Should set the `ADMIN` as the admin of `CONFIG_ADMIN`.
  - ☑ Test coverage
- Should set the `ADMIN` as the admin of `UPDATER_ADMIN`.
  - ☑ Test coverage
- Should set the `UPDATER_ADMIN` as the admin of `ORACLE_UPDATER`.
  - ☑ Test coverage

Negative behavior

- Should not be callable multiple times. That is enforced as it is an internal function, only called in the constructor.
  - ☑ Negative test
- Should not allow anyone other than the `ADMIN` to call this function. Technically enforced as it is an internal function, only called in the constructor.
  - ☑ Negative test

Function: `setConfig(Config newConfig)`

This allows admin to change the accumulator configuration.

Inputs

- `newConfig`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked that `updateDelay` is not greater than heartbeat and that `updateThreshold` is not zero.
    - **Impact**: The new configuration to be set.

Branches and code coverage

Intended branches

- Assure that the `newConfig.heartbeat` is not zero. Currently not checking for this.
    - ☐ Test coverage
- Should set the new configuration.
    - ☑ Test coverage
- Assured that the new configuration's `updateDelay` is not greater than heartbeat.
    - ☑ Test coverage
- Assure that the new configuration's `updateThreshold` is not zero.
    - ☑ Test coverage

Negative behavior

- Should not allow setting the same configuration again. Currently, not checking for this.
    - ☐ Negative test
- Should not allow anyone other than the `CONFIG_ADMIN` to call this function.
    - ☑ Negative test

5.7. Module: AccumulatorOracleView.sol

Function: `getLatestObservation(address token)`

This returns the latest observation from observation data by consulting the underlying accumulators.

Inputs

- `data`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.

- **Impact**: The token for which observation is returned.

Branches and code coverage

Intended branches

- Call the underlying accumulators to consult the price and liquidity and uses the oldest of the two timestamps as the observation timestamp.
  - ☑ Test coverage

Negative behavior

- N/A.

Function call analysis

- `PriceAccumulator(priceAccumulator).lastUpdateTime(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the last update time. Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `LiquidityAccumulator(liquidityAccumulator).lastUpdateTime(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the last update time. Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `PriceAccumulator(priceAccumulator).consultPrice(token)`
  - **What is controllable?** `token`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the latest price from the accumulator. Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `PriceAccumulator(priceAccumulator).consultPrice(token)`
  - **What is controllable?** `token`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the latest liquidities of tokens and quote tokens from the accumulator. Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

5.8. Module: CapController.sol

Function: `setChangeThreshold(address token, uint32 changeThreshold)`

This allows setting the change threshold for a token.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. Assumes the token has already been added to the rate controller.
  - **Impact**: The token must be added to the rate controller before calling this function.
- `changeThreshold`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The change threshold.

Branches and code coverage

Intended branches

- Set the rate buffer metadata for the token.
  - ☑ Test coverage
- Only update the rate buffer metadata if the change threshold is different.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the `RATE_ADMIN`. Enforced on each `check-SetChangeThreshold` individual implementation.
  - ☑ Negative test

5.9. Module: ChainlinkOracleView.sol

Function: `getLatestObservation(address token)`

This returns the latest observation from observation data by consulting the underlying price feed.

Inputs

- `data`
  - **Control**: Fully controlled by the caller.

- **Constraints**: No constraints.
- **Impact**: The token for which observation is returned.

Branches and code coverage

Intended branches

- Call the underlying price feed to consult the price.
  - ☑ Test coverage

Negative behavior

- Revert if token is not the `feedToken`.
  - ☑ Negative test
- Revert if the price returned by the feed is negative.
  - ☑ Negative test
- Revert if the price returned by the feed is greater than `type(uint112).max`.
  - ☑ Negative test
- Revert if `updatedAt` is zero or greater than `type(uint32).max`.
  - ☑ Negative test

Function call analysis

- `AggregatorV3Interface(getUnderlyingFeed()).latestRoundData()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the price and last time price feed was updated. Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

5.10. Module: CurrentAggregatorOracle.sol

Function: `update(bytes data)`

This function aggregates the prices from the underlying oracles and pushes the prices in observation buffers.

Inputs

- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: Encoded data for the oracle.

Branches and code coverage

Intended branches

- Call `performUpdate` if `needsUpdate` returns true.
  - ☑ Test coverage
- The function `needsUpdate` returns true if the time since last update is greater than or equal to `_heartbeat()` or the price has changed by a certain threshold if time since last update is greater than `_updateDelay()` and less than `_heartbeat()`.
  - ☑ Test coverage
- The function pushes the aggregated observation if the number of valid observations is greater than `_minimumResponses(token)`.
  - ☑ Test coverage

Negative behavior

- Return without calling `performUpdate` if `needsUpdate` returns false.
  - ☑ Negative test
- The function `needsUpdate` returns false if the time since last update is less than `_updateDelay()` or price has not changed by certain threshold.
  - ☑ Negative test

Function call analysis

- `this.needsUpdate(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true or false. If true, the function performs an update; otherwise, it returns.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.updateUnderlyingOracles(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if at least one of the oracles is updated.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.updateUnderlyingOracles(data) -> IOracle(theOracles[i].oracle).update(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true or false. If true, the function performs an update; otherwise, it returns.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the function `updateUnderlyingOracles` will catch the error and execute the remaining code.

- `this.performUpdate(data) -> this.aggregateUnderlying(token,_maximumResponseAge(token))`
    - **What is controllable?** `token`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns number of valid observations and the aggregated result.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.aggregateUnderlying(token,_maximumResponseAge(token)) -> IOracle(theOracles[i].oracle).consult(token, maxAge)`
    - **What is controllable?** `token`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the price and liquidity from the underlying oracle.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the function `aggregateUnderlying` will catch the error and execute the remaining code.
- `this.performUpdate(data) -> this.aggregateUnderlying(token,_maximumResponseAge(token)) -> IOracle(theOracles[i].oracle).lastUpdateTime(updateData)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the last time the oracle was updated.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.aggregateUnderlying(token,_maximumResponseAge(token)) -> validation.validateObservation(token, observation)`
    - **What is controllable?** `token`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if observation is valid, false otherwise.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.aggregateUnderlying(token,_maximumResponseAge(token)) -> _aggregationStrategy(token).aggregateObservations(token, observations, 0, validResponses - 1)`
    - **What is controllable?** `token`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the aggregated value of the observation.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

5.11. Module: HistoricalAggregatorOracle.sol

Function: `getLatestObservation(address token)`

This returns the latest observation from the observation buffer.

Inputs

- `data`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The token for which observation is returned.

Branches and code coverage

Intended branches

- Return empty buffer if `meta.size` is zero.
    - ☑ Test coverage

Negative behavior

- N/A.

5.12. Module: HistoricalOracle.sol

Function: `setObservationsCapacity(address token, uint256 amount)`

This function is used to set the `maxSize` of the observation buffer.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The token for which `maxSize` would be set.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be greater than or equal to the current `maxSize` but less than 65,536.
    - **Impact**: The new capacity of observations for the token.

Branches and code coverage

Intended branches

- If `meta.maxSize` of the token is zero, initialize the buffer.

Negative behavior

- Revert if the new amount is less than the current capacity or greater than 65,535.

5.13. Module: HistoricalRates.sol

Function: `setRatesCapacity(address token, uint256 amount)`

This allows setting the rate-buffer capacity for a token.

Inputs

- `token`
  - **Control**: The token for which to set the new capacity.
  - **Constraints**: Assumed to exist — not explicitly checked.
  - **Impact**: The token for which to set the new capacity.
- `amount`
  - **Control**: The new capacity of rates for the token. Must be greater than the current capacity but less than 256.
  - **Constraints**: Must be greater than the current capacity but less than 256.
  - **Impact**: The new capacity of rates for the token.

Branches and code coverage

Intended branches

- Check that `amount` is greater than `meta.maxSize`.
  - ☑ Test coverage
- Check that `amount` is less than `type(uint16).max`.
  - ☑ Test coverage
- Check that `meta.maxSize` is not equal to `amount`.
  - ☑ Test coverage
- Push dummy values to the buffer in the case that `meta.maxSize` is not equal to `amount`.
  - ☑ Test coverage
- Update `meta.maxSize` to `amount`.
  - ☑ Test coverage

5.14. Module: LiquidityAccumulator.sol

Function: `consultLiquidity(address token)`

This returns the token and quote-token liquidity from the observation mapping.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The token for which liquidity is returned.

Branches and code coverage

Intended branches

- Return `0,0` if `token` is a quote token.
  - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test

Function: `consultLiquidity(address token, uint256 maxAge)`

This returns the token and quote-token liquidity from the observation mapping. The maximum age of observation is bound to `maxAge`

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The token for which liquidity is returned.
- `maxAge`
  - **Control**: Fully controlled by caller.
  - **Constraints**: No constraints.
  - **Impact**: The maximum age of the quotation, in seconds. If zero, fetches the real-time liquidity.

Branches and code coverage

Intended branches

- Return `0,0` if `token` is a quote token.
  - ☑ Test coverage
- Call `fetchLiquidity` to fetch the real-time liquidity if `maxAge` is zero.
  - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test
- Revert if `observation.timestamp + maxAge` is less than `block.timestamp`.
  - ☑ Negative test

Function: `update(bytes data)`

This updates the accumulator for a specific token.

Inputs

- `data`
  - **Control**: Fully controlled by caller.
  - **Constraints**: No constraints.
  - **Impact**: Encoding of the token address followed by the expected token liquidity and quote-token liquidity.

Branches and code coverage

Intended branches

- Call `performUpdate` if `needsUpdate` returns true.
  - ☑ Test coverage
- The function `needsUpdate` returns true if the time since last update is greater than or equal to `_heartbeat()` or liquidity has changed by a certain threshold if the time since last update is greater than `_updateDelay()` and less than `_heartbeat()`.
  - ☑ Test coverage

Negative behavior

- Return without calling `performUpdate` if `needsUpdate` returns false.
  - ☑ Negative test
- The function `needsUpdate` returns false if the time since last update is less than `_updateDelay()` or liquidity has not changed by a certain threshold.
  - ☑ Negative test

- The function `performUpdate` returns false if the observation is invalid.
  - ☑ Negative test

Function call analysis

- `this.needsUpdate(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true or false. If true, the function performs an update; otherwise, it returns.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.fetchLiquidity(data)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the liquidity of tokens and quote tokens.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.validateObservation(data, tokenLiquidity, quoteTokenLiquidity)`
  - **What is controllable?** `data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if observation is valid, false otherwise.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

5.15. Module: ManagedAaveV2RateAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)`(i.e., anyone) have the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

Performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)`(i.e., anyone) have the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.16. Module: ManagedAaveV3RateAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)`(i.e anyone) have the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)`(i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.17. Module: ManagedAaveV3SBAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORA-CLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

5.18. Module: ManagedAggregatorOracleBase.sol

Function: `setTokenConfig(address token, IOracleAggregatorTokenConfig newConfig)`

This updates the configuration of a specific token.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The token to set the configuration for.
- `newConfig`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The new configuration for the token.

Branches and code coverage

Intended branches

- Assume no malicious intent when calling this function, as the `newConfig` is fully controlled by the caller.
  - ☑ Test coverage
- Update the `tokenConfigs` mapping of the token to the new configuration.
  - ☑ Test coverage
- Assume that the validation strategy and aggregation strategy are legitimate. This is not explicitly checked (through some whitelist, for example).
  - ☑ Test coverage

Negative behavior

- Revert if not enough oracles are provided.
  - ☑ Negative test
- Revert if the minimum responses are zero.
  - ☑ Negative test
- Should not be callable by anyone other than the `CONFIG_ADMIN` role.
  - ☑ Negative test
- Revert if oracles are not unique. This could be made more gas efficient by using a mapping instead of a nested loop.
  - ☑ Negative test
- Revert if the validation strategy's quote token decimals do not match the aggregator's quote token decimals.
  - ☑ Negative test

Function call analysis

- `newConfig.oracles()`
  - **What is controllable?** `newConfig`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is the oracles of the configuration. These could be arbitrary addresses. Could be malicious.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `newConfig.minimumResponses()`
  - **What is controllable?** `newConfig`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is the minimum responses of the configuration. This could be an arbitrary number.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `newConfig.aggregationStrategy()`

  - **What is controllable?** `newConfig`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is the aggregation strategy of the configuration. This could be an arbitrary address. Could be malicious.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `newConfig.validationStrategy()`
  - **What is controllable?** `newConfig`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is the validation strategy of the configuration. This could be an arbitrary address. Could be malicious.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `validationStrategy.quoteTokenDecimals()`
  - **What is controllable?** `validationStrategy` through `newConfig`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is the quote token decimals of the validation strategy. This could be an arbitrary number.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.19. Module: ManagedAlgebraLiquidityAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.20. Module: ManagedAlgebraPriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.

- **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
- **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assumes that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.21. Module: ManagedBalancerV2LiquidityAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.22. Module: ManagedBalancerV2StablePriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.23. Module: ManagedBalancerV2WeightedPriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.

- ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.24. Module: ManagedCometRateAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.25. Module: ManagedCometSBAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.26. Module: ManagedCompoundV2RateAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

5.27. Module: ManagedCurrentAggregatorOracleBase.sol

Function: `setConfig(Config newConfig)`

Allows admin to change the accumulator configuration,

Inputs

- `newConfig`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked that `updateDelay` is not greater than heartbeat, that `updateThreshold` is not zero, and that heartbeat is not zero.
    - **Impact**: The new configuration to be set.

Branches and code coverage

Intended branches

- Assure that the `newConfig.heartbeat` is not zero.
  - ☑ Test coverage
- Should set the new configuration.
  - ☑ Test coverage
- Assure that the new configuration's `updateDelay` is not greater than heartbeat.
  - ☑ Test coverage
- Assure that the new configuration's `updateThreshold` is not zero.
  - ☑ Test coverage

Negative behavior

- Should not allow setting the same configuration again. Currently, not checking for this.
  - ☐ Negative test
- Should not allow anyone other than the `CONFIG_ADMIN` to call this function.
  - ☑ Negative test

5.28. Module: ManagedCurrentAggregatorOracle.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `setUpdatesPaused(address token, bool paused)`

This pauses/unpauses the updates of the oracle for the given token.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed it is a valid token address.
    - **Impact**: The token to pause/unpause updates for.
- `paused`
    - **Control**: The value to set the pause flag to.
    - **Constraints**: None.
    - **Impact**: The value to set the pause flag to.

Branches and code coverage

Intended branches

- If `paused` is true, set the pause flag to true.
    - ☑ Test coverage
- If `paused` is false, set the pause flag to false.
    - ☑ Test coverage
- Update the pause flag for the given token.
    - ☑ Test coverage

Negative behavior

- Nobody other than the admin should be able to call this function.
    - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the oracle.

Inputs

- `data`
    - **Control**: The data to update the oracle with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.29. Module: ManagedCurveLiquidityAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.30. Module: ManagedCurvePriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.31. Module: ManagedHistoricalAggregatorOracleBase.sol

Function: `setConfig(Config newConfig)`

This allows the admin to change the accumulator configuration.

Inputs

- `newConfig`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked that `updateDelay` is not greater than heartbeat and that `updateThreshold` is not zero.
    - **Impact**: The new configuration to be set.

Branches and code coverage

Intended branches

- Update the `config` to the `newConfig`.
    - ☑ Test coverage
- Assure that the new configuration's `observationAmount` is not zero.
    - ☑ Test coverage
- Assure that the new configuration's `observationIncrement` is not zero.
    - ☑ Test coverage
- Assure that the new configuration's `source` is not the zero address.
    - ☑ Test coverage
- Assure that the new configuration's decimals match the token decimals of the current configuration.
    - ☑ Test coverage

Negative behavior

- Should not allow setting the same configuration again. Currently, not checking for this.
    - ☐ Negative test
- Should not allow anyone other than the `CONFIG_ADMIN` to call this function.
    - ☑ Negative test

5.32. Module: ManagedMedianFilteringOracle.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `setUpdatesPaused(address token, bool paused)`

This pauses/unpauses the updates of the oracle for the given token.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumed it is a valid token address.
  - **Impact**: The token to pause/unpause updates for.
- `paused`
  - **Control**: The value to set the pause flag to.
  - **Constraints**: None.
  - **Impact**: The value to set the pause flag to.

Branches and code coverage

Intended branches

- If `paused` is true, set the pause flag to true.
  - ☑ Test coverage
- If `paused` is false, set the pause flag to false.
  - ☑ Test coverage

- Update the pause flag for the given token.
  - ☑ Test coverage

Negative behavior

- Nobody other than the admin should be able to call this function.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.33. Module: ManagedOffchainLiquidityAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that `msg.sender` has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow calling this function unless `msg.sender` has the `ORACLE_UPDATER` role.

☑ Negative test

5.34. Module: ManagedOffchainPriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)`(i.e., anyone)

have the `ORACLE_UPDATER` role.

- **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  ☑ Test coverage

Negative behavior

- Should not allow calling this function unless `msg.sender` has the `ORACLE_UPDATER` role.
  ☑ Negative test

5.35. Module: ManagedOracleBase.sol

Function: `initializeRoles()`

This initializes the roles for the contract.

Branches and code coverage

Intended branches

- Should set the deployer as the admin.
  ☑ Test coverage
- Should set the `ADMIN` as the admin of `CONFIG_ADMIN`.
  ☑ Test coverage

- Should set the `ADMIN` as the admin of `UPDATER_ADMIN`.
  - ☑ Test coverage
- Should set the `UPDATER_ADMIN` as the admin of `ORACLE_UPDATER`.
  - ☑ Test coverage

Negative behavior

- Should not be callable multiple times. That is enforced as it is an internal function, only called in the constructor.
  - ☑ Negative test
- Should not allow anyone other than the ADMIN to call this function. Technically enforced as it is an internal function, only called in the constructor.
  - ☑ Negative test

5.36. Module: ManagedPeriodicAccumulationOracle.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.37. Module: ManagedPeriodicAggregatorOracle.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.

    ☑ Test coverage

Negative behavior

- Should not allow anyone to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
    ☑ Negative test

Function: `setUpdatesPaused(address token, bool paused)`

This pauses/unpauses the updates of the oracle for the given token.

Inputs

- `token`
    - **Control**: Fully controlled by caller.
    - **Constraints**: Assumed it is a valid token address.
    - **Impact**: The token to pause/unpause updates for.
- `paused`
    - **Control**: The value to set the pause flag to.
    - **Constraints**: None.
    - **Impact**: The value to set the pause flag to.

Branches and code coverage

Intended branches

- If `paused` is true, set the pause flag to true.
    ☑ Test coverage
- If `paused` is false, set the pause flag to false.
    ☑ Test coverage
- Update the pause flag for the given token.
    ☑ Test coverage

Negative behavior

- Nobody other than the admin should be able to call this function.
    ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the oracle.

Inputs

- `data`
    - **Control**: The data to update the oracle with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

5.38. Module: ManagedPeriodicPriceAccumulationOracle.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the oracle.

Inputs

- `data`
    - **Control**: The data to update the oracle with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.

- ☑ Test coverage

Negative behavior

- Should not allow anyone to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.39. Module: ManagedPriceVolatilityOracle.sol

Function: `canUpdate(byte[] data)`

Determines whether `msg.sender` can update the oracle.

Inputs

- `data`
    - **Control**: The data to update the oracle with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)`(i.e anyone) have the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

Function: `setUpdatesPaused(address token, bool paused)`

This pauses/unpauses the updates of the oracle for the given token.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed it is a valid token address.
    - **Impact**: The token to pause/unpause updates for.
- `paused`
    - **Control**: The value to set the pause flag to.
    - **Constraints**: None.
    - **Impact**: The value to set the pause flag to.

Branches and code coverage

Intended branches

- If `paused` is true, set the pause flag to true.
    - ☑ Test coverage
- If `paused` is false, set the pause flag to false.
    - ☑ Test coverage

- Update the pause flag for the given token.
  - ☑ Test coverage

Negative behavior

- Nobody other than the admin should be able to call this function.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the oracle.

Inputs

- `data`
  - **Control**: The data to update the oracle with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the oracle, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.40. Module: ManagedUniswapV2LiquidityAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.41. Module: ManagedUniswapV2PriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
  - ☑ Test coverage
- Should call `super.update(data)`.
  - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

5.42. Module: ManagedUniswapV3LiquidityAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
  - **Control**: The data to update the accumulator with.
  - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
  - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

5.43. Module: ManagedUniswapV3PriceAccumulator.sol

Function: `canUpdate(byte[] data)`

This determines whether `msg.sender` can update the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Checks performed on the data to be updated.

Branches and code coverage

Intended branches

- Should be called within `update()`.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the accumulator.

Inputs

- `data`
    - **Control**: The data to update the accumulator with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the accumulator, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

5.44. Module: ManualRateComputer.sol

Function: `setRate(address token, uint64 rate)`

This allows manually setting the rate of a token.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed to be a valid token address. Not enforced.
    - **Impact**: The token whose rate is being set.
- `rate`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed to be a valid rate. Not enforced.
    - **Impact**: The rate to set for the token.

Branches and code coverage

Intended branches

- Set the rate for the token.
    - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the `RATE_ADMIN`. Enforced on each `check-SetRate` individual implementation.
    - ☑ Negative test

5.45. Module: MutatedValueComputer.sol

Function: `setConfig(address token, uint64 max, uint64 min, int64 off-set, uint32 scalar)`

This allows owner to change the configuration of a token.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be a valid token address.
    - **Impact**: The token whose configuration is being changed.

- `max`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be a valid `uint64`.
    - **Impact**: The maximum value for the token's rate.
- `min`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be a valid `uint64`.
    - **Impact**: The minimum value for the token's rate.
- `offset`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be a valid `int64`.
    - **Impact**: The offset to apply to the computed value.
- `scalar`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be a valid `uint32`.
    - **Impact**: The scalar value to apply to the computed value.

Branches and code coverage

Intended branches

- In the case that the config already exists for the token, assumed that the max config is greater than the current rate value. This is not enforced.
    - ☐ Test coverage
- In the case that the config already exists for the token, assumed that the min config is less than the current rate value. This is not enforced.
    - ☐ Test coverage
- Assure that `scalar` is different than zero.
    - ☑ Test coverage
- Set the new config for the token with the given values.
    - ☑ Test coverage

Negative behavior

- Should not allow min to be greater than max.
    - ☑ Negative test
- Should only allow the service admin to set the config for a token. Enforced through the `checkSetConfig` function.
    - ☑ Negative test

5.46. Module: PeriodicAccumulationOracle.sol

Function: `getLatestObservation(address token)`

This returns the latest observation from the observation buffer.

Inputs

- `data`
    - **Control**: Fully controlled by caller.
    - **Constraints**: No constraints.
    - **Impact**: The token for which the observation is returned.

Branches and code coverage

Intended branches

- Return the observation from `observations` mapping.
    - ☑ Test coverage

Negative behavior

- N/A.

5.47. Module: PriceAccumulator.sol

Function: `consultPrice(address token)`

This returns the price from the observation mapping.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.
    - **Impact**: The token for which price is returned.

Branches and code coverage

Intended branches

- Return `10 ** quoteTokenDecimals()` if `token` is a quote token.
    - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test

Function: `consultPrice(address token, uint256 maxAge)`

This returns the price from the observation mapping. The maximum age of observation is bound to `maxAge`.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The token for which the price is returned.
- `maxAge`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: The maximum age of the quotation, in seconds. If zero, fetches the real-time price.

Branches and code coverage

Intended branches

- Return `10 ** quoteTokenDecimals()` if `token` is a quote token.
  - ☑ Test coverage
- Call `fetchPrice` to fetch the real-time price if `maxAge` is zero.
  - ☑ Test coverage

Negative behavior

- Revert if timestamp of the observation is zero.
  - ☑ Negative test
- Revert if `observation.timestamp + maxAge` is less than `block.timestamp`.
  - ☑ Negative test

Function: `update(bytes data)`

This updates the accumulator for a specific token.

Inputs

- `data`
    - **Control**: Fully controlled by caller.
    - **Constraints**: No constraints.
    - **Impact**: Encoding of the token address followed by the expected price.

Branches and code coverage

Intended branches

- Call `performUpdate` if `needsUpdate` returns true.
    - ☑ Test coverage
- The function `needsUpdate` returns true if the time since last update is greater than or equal to `_heartbeat()` or the price has changed by a certain threshold if the time since last update is greater than `_updateDelay()` and less than `_heartbeat()`.
    - ☑ Test coverage

Negative behavior

- Return without calling `performUpdate` if `needsUpdate` returns false.
    - ☑ Negative test
- The function `needsUpdate` returns false if the time since the last update is less than `_updateDelay()` or price has not changed by certain threshold.
    - ☑ Negative test
- The function `performUpdate` returns false if observation is invalid.
    - ☑ Negative test

Function call analysis

- `this.needsUpdate(data)`
    - **What is controllable?** `data`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns true or false. If true, the function performs an update; otherwise, it returns.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.fetchPrice(data)`
    - **What is controllable?** `data`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the price of the token.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.
- `this.performUpdate(data) -> this.validateObservation(data, price)`
    - **What is controllable?** `data`.

- **If the return value is controllable, how is it used and how can it go wrong?** Returns true if observation is valid, false otherwise.
- **What happens if it reverts, reenters or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenario.

5.48. Module: RateController.sol

Function: `manuallyPushRate(address token, uint64 target, uint64 current, uint256 amount)`

This allows manually pushing a rate to the rate buffer.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked that a `rateBufferMetadata` exists for the token (i.e., the token has a rate config).
    - **Impact**: The token whose rate is being pushed.
- `target`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed to be a valid target rate. Not enforced.
    - **Impact**: The target rate to push.
- `current`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed to be a valid current rate. Not enforced.
    - **Impact**: The current rate to push.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed to be a valid amount.
    - **Impact**: The number of times to push the rate.

Branches and code coverage

Intended branches

- Push the rate to the buffer.
    - [x] Test coverage
- Assure that the buffer is initialized (i.e., the rate config exists for the given token).
    - [x] Test coverage

Negative behavior

- Should not be callable by anyone other than the admin. This is enforced by the `checkManuallyPushRate` function in inheriting contracts.
  - ☑ Negative test

Function: `setConfig(address token, RateConfig config)`

This allows setting the rate configuration of a token.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumed to be a valid token address. Not enforced.
  - **Impact**: The token whose rate configuration is being set.
- `config`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Multiple checks on all relevant parameters of the config are performed.
  - **Impact**: The rate configuration to set.

Branches and code coverage

Intended branches

- Assure that each individual `componentWeight` is not zero. Currently not performed.
  - ☐ Test coverage
- Assure that the sum of `componentWeights` is not zero. Currently not performed.
  - ☐ Test coverage
- Assure that the `config.components.length == config.componentWeights.length`.
  - ☑ Test coverage
- Assure that the `config.maxPercentDecrease <= 10000`.
  - ☑ Test coverage
- Assure that the `config.max >= config.min`.
  - ☑ Test coverage
- Assure that the component weights sum to less than or equal to 10,000.
  - ☑ Test coverage
- Ensure that the base rate plus the sum of the maximum component rates does not overflow.
  - ☑ Test coverage
- Initialize the buffers if they are not already initialized.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin. This is enforced by the `checkSetConfig` function in inheriting contracts.
  - ☑ Negative test

Function: `setUpdatesPaused(address token, bool paused)`

This allows pausing/unpausing rate updates for a token.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the rate buffer metadata exists for the token (i.e., the token has a rate config).
  - **Impact**: The token whose rate updates are being paused/unpaused.
- `paused`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Whether rate updates should be paused/unpaused.

Branches and code coverage

Intended branches

- Check that `rateBufferMetadata` exists for the token (i.e., the token has a rate config).
  - ☑ Test coverage
- If `paused` is true, set the pause flag to true.
  - ☑ Test coverage
- If `paused` is false, set the pause flag to false.
  - ☑ Test coverage

Negative behavior

- Should not update the pause flag if it is already set to the desired value. Not performed.
  - ☐ Negative test
- Should not be callable by anyone other than the admin. This is enforced by the `checkSetUpdatesPaused` function in inheriting contracts.
  - ☑ Negative test

Function: `update(byte[] data)`

This performs an update on the rate controller.

Inputs

- `data`
    - **Control**: The data to update the rate controller with.
    - **Constraints**: Checks that either `msg.sender` or `address(0)` (i.e., anyone) has the `ORACLE_UPDATER` role.
    - **Impact**: Updates the accumulator with the data.

Branches and code coverage

Intended branches

- Should use the `onlyRoleOrOpenRole` modifier.
    - ☑ Test coverage
- Should call `super.update(data)`.
    - ☑ Test coverage
- Assume that `super.update` performs the necessary checks.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `ORACLE_UPDATER` to update the rate controller, unless `address(0)` has the `ORACLE_UPDATER` role.
    - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was partly deployed to the Polygon mainnet.

During our assessment on the scoped Adrastia Protocol contracts, we discovered six findings. No critical issues were found. Four findings were of medium impact and two were of low impact. Adrastia acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.