Security Review Report
NM-0081 Ondo Finance

NETHERMIND

(April 6, 2023)

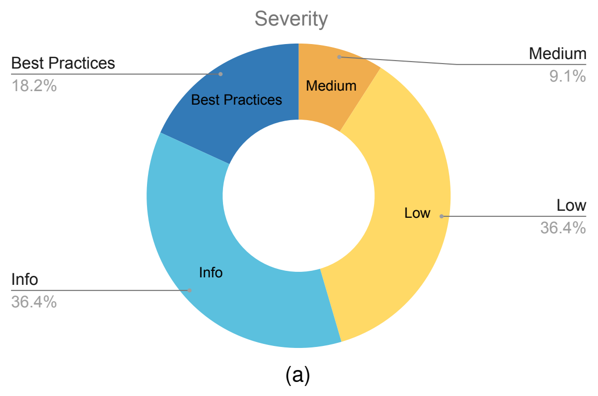
Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind for the Ondo Finance. The audit primarily focuses on oracles' pricing assets on the Flux and CASH protocols. `Flux Finance` is a decentralized lending protocol developed by the Ondo Finance team that supports both permissionless and permissioned tokens. Assets on Flux are represented as `fTokens`. The `CASH` protocol, on the other hand, enables whitelisted users to gain exposure to RWAs through Cash tokens. The protocol employs Know Your Customer (KYC) procedures to allow users to mint Cash tokens.

The audited code consists of 451 lines of Solidity. The Ondo Finance team provided extensive documentation in the audit repository. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. **Along this document, we report** 11 points of attention, where one is classified as `Medium`, four are classified as `Low`, four are classified as `Informational` and two are classified as `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



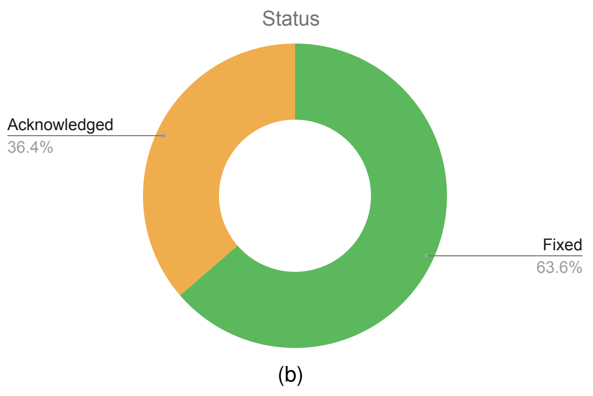


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (4), **Undetermined** (0), **Informational** (4), **Best Practices** (2).
Distribution of status: Fixed (7), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr. 19, 2023
Response from Client	Apr. 21, 2023
Final Report	Apr. 24, 2023
Methods	Manual Review, Automated Analysis
Repository	Ondo
Commit Hash (Initial Audit)	6face44a82b7300be928cba30cd970636a77d93a
Documentation	README.md
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/lending/fluxOracles/FluxOracle.sol	190	138	72.6%	31	359
2	contracts/lending/rwaOracles/RWAOracleRateCheck.sol	52	60	115.4%	19	131
3	contracts/lending/rwaOracles/RWAOracleExternalComparisonCheck.sol	135	91	67.4%	30	256
4	contracts/lending/fTokenOracle/fTokenOracle.sol	74	57	77.0%	17	148
	Total	**451**	**346**	**76.7%**	**97**	**894**

3 Summary of Issues

	Finding	Severity	Update
1	Rounding issue when calculating absolute changes in basis points (BPS)	Medium	Acknowledged
2	Changing oracle type does not remove previous oracle data	Low	Fixed
3	Lack of a two-step process for transferring ownership	Low	Fixed
4	Use AccessControlDefaultAdminRules	Low	Acknowledged
5	`_setFTokenToChainlinkOracle(...)` does not check oracle decimals	Low	Acknowledged
6	Events in RWA oracles differ	Info	Fixed
7	Inconsistent behavior on `price == 0`	Info	Fixed
8	Incorrect check for decimals	Info	Fixed
9	No check for fToken validity	Info	Fixed
10	Use of magic numbers	Best Practices	Fixed
11	`renounceOwnership(...)` function should be disabled	Best Practices	Acknowledged

4 System Overview

The audit is based on four contracts: a) `FluxOracle`; b) `RWAOracleRateCheck`; c) `RWAOracleExternalComparisonCheck`; and, d) `fTokenOracle`. Fig.1 illustrates a structural diagram of the contracts and their association.

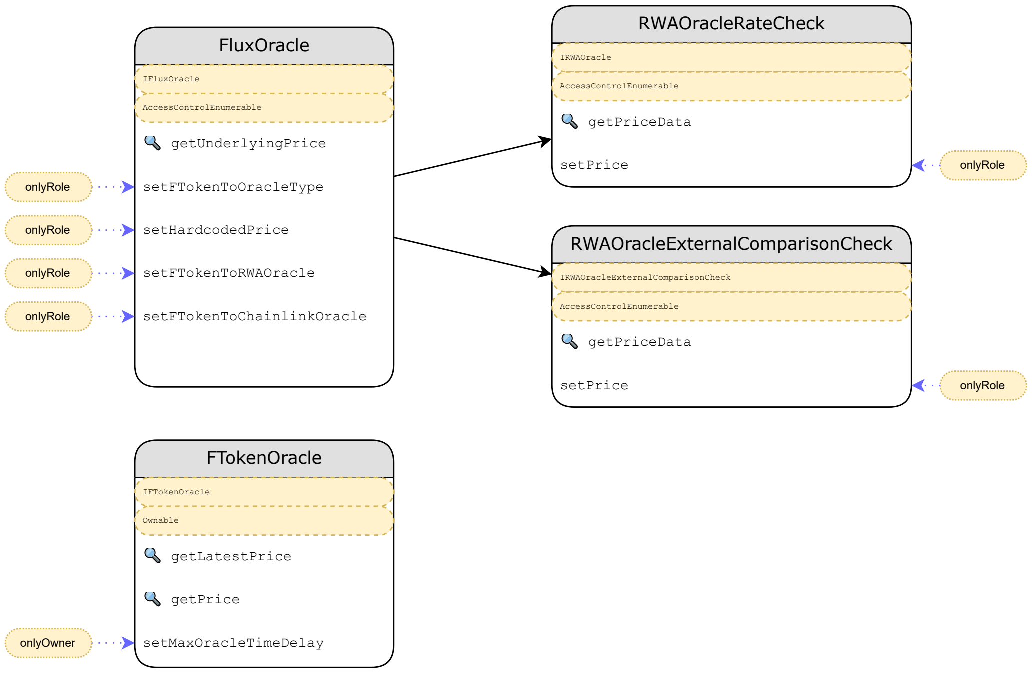


Figure 1: Structural Diagram of Ondo Oracles

The **FluxOracle** contract functions as a price oracle for the lending market, and it supports three distinct oracle types: `Hardcoded`, `Tokenized_RWA`, and `Chainlink`. The `DEFAULT_ADMIN_ROLE` is the only role that can set an asset's oracle type, and each oracle type has a specific role for updating its parameters. When retrieving the price of a `Tokenized_RWA` asset, the contract makes an external call to an oracle that implements the `IRWAOracle` interface. For assets with `Chainlink` oracle type, the contract makes an external call to a `Chainlink` price feed, which returns the asset's price in USD with the price expressed in 36 decimals of the underlying asset. The contract uses the struct `ChainlinkOracleInfo`:

```
struct ChainlinkOracleInfo {
    AggregatorV3Interface oracle;
    uint256 scaleFactor;
    uint256 maxChainlinkOracleTimeDelay;
}
```

The contract features several public functions

- `getUnderlyingPrice(...)` is used to retrieve the price based on the associated `OracleType`
- `setFTokenToOracleType(...)` is used to set the oracle type for the provided `fToken`
- `setHardcodedPrice(...)` is used to set the price of an `fToken` contract's underlying asset
- `setFTokenToRWAOracle(...)` is used to associate a custom `fToken` with an external `cToken`
- `setFTokenToChainlinkOracle(...)` is used to associate a custom `fToken` with a Chainlink oracle

The **RWAOracleRateCheck** contract implements the `IRWAOracle` interface to determine the price of RWAs, such as CASH tokens. It employs a trusted off-chain party to set the price of a token within a specified `MAX_CHANGE_DIFF_BPS` limit of 100 basis points or 1%. In other words, the contract ensures a secure pricing mechanism for RWAs, which are critical assets on the Flux platform, by permitting a trusted source to determine their value within a predetermined range. The contract features two public functions:

- `getPriceData(...)` is used to retrieve the most recently set RWA price data.
- `setPrice(...)` is used by `SETTER_ROLE` to set a new RWA price. It will verify that the previous price update was not done too recently and that the proposed price change does not exceed 1

The **RWAOracleExternalComparisonCheck** contract is designed to price RWAs through the implementation of the `IRWAOracle` interface. This Oracle contract ensures the accuracy of RWA prices by cross-verifying them with a `Chainlink` price feed and a trusted 3rd-party. In the case of OUSG, the price is verified with Chainlink's SHV Oracle. The contract applies several checks before setting the price. These include verifying that the RWA price hasn't been updated too recently, the difference between the old and new RWA price isn't greater than `MAX_ABSOLUTE_DIFF_BPS`, the difference between the old and new Chainlink price isn't greater than `MAX_ABSOLUTE_DIFF_BPS + MAX_-CHANGE_DIFF_BPS` (otherwise the Chainlink price is flagged as malfunctioning), and the change in RWA price hasn't deviated more than `MAX_CHANGE_DIFF_BPS` from the change in the Chainlink price. Essentially, the `RWAOracleExternalComparisonCheck` provides an additional layer of security by ensuring the accuracy of RWA prices through cross-checking them with a trusted third-party and a `Chainlink` price feed. The contract features two public functions:

– `getPriceData(...)` is used to retrieve the most recently set RWA price data.

– `setPrice(...)` is used by `SETTER_ROLE` to set a new RWA price. This function applies the same checks as those in the `RWAOracleRateCheck` and additionally incorporates a cross-check with a Chainlink price feed to confirm the accuracy of the RWA prices that are set by a trusted third-party.

The **fTokenOracle** contract serves as an Oracle to determine the price of an `fToken`. This is achieved by using the exchange rate of `fToken` to its underlying asset in the Flux lending market and multiplying it with the price feed of the underlying asset in USD from Chainlink. The resulting calculation produces the price of the `fToken` in USD with 18 decimal places. The `fTokenOracle` provides valuable pricing information for `fToken`, which are tokens representing interest-bearing claims on underlying assets. The contract features several public functions

– `getLatestPrice(...)` is utilized to retrieve a non-stale price of `fToken` denominated in the asset specified by `underlyingPriceFeed`. This function acquires the price from the Chainlink price feed and applies additional checks to the returned price to confirm that it is not stale.

– `getPrice(...)` is similar to `getLatestPrice(...)`, but it does not include the additional checks. Therefore, the price returned by this function may be stale.

– `setMaxOracleTimeDelay(...)` is used by the owner to set the maximum time delay for the Chainlink price feed.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Rounding issue when calculating absolute changes in basis points (BPS)

File(s): contracts/lending/rwaOracles/RWAOracleExternalComparisonCheck.sol, contracts/lending/rwaOracles/RWAOracleRateCheck.sol

Description: In the RWA oracle, there is a MAX_CHANGE_DIFF_BPS parameter that limits the change in the new price. This is done by calculating the difference between the new and old prices and dividing it by the old price. However, rounding down when calculating changeBps could allow price change to be higher than MAX_CHANGE_DIFF_BPS.

Example of an exploited scenario:

1. The admin sets MAX_CHANGE_DIFF_BPS = 10;
2. If the old price is previousPrice = 10011 and the new price is newPrice = 10000, the calculation will be:

```
change = newPrice - previousPrice = 10000 - 10011 = -11
changeBps = (change * BPS_DENOMINATOR) / previousPrice = (-11 * 10000) / 10011 = -10.99
```

3. Since Solidity performs calculations using integers, changeBps will round down to -10, making it still valid to set even though the change is more than 10 BPS;

Recommendation(s): Consider rounding up when calculating the absolute changes in basis points. Here is a reference implementation for rounding up.

Status: Acknowledged

Update from the client: This is a great call out, but exclusively rounding up or down would have a loosening or tightening effect based on the situation. The slightly more correct option here would be to conditionally round up or down depending on which action would result in the tightest possible constraint. Due to how loose this constraint already is, we do not expect to ever be operating at these bounds; this decision should have no tangible outcome of the goals of the contract. We have decided not to implement “conditional” rounding as it is not worth the complexity.

Example where rounding down loosens constraint. If RWA bps change is 84.9, and the CL bps change is 10, the 84.9 gets truncated to 84. Because 84-10 = 74, the price setting would succeed.

Example where rounding down tightens constraint If RWA bps change is -85 and the CL bps change is -10.9, The -10.9 gets truncated to -10. Because -85 - -10 = -75, the price setting would fail. (If we rounded up it would have “loosened” constraint and passed)

Update from Nethermind: We agree with your perspective that rounding should be situation-dependent. However, we believe it remains an issue, not a false positive. The provided examples occur because RWAOracleExternalComparisonCheck employs two rounds of rounding (one for RWA BPS and one for Chainlink BPS), while in the case of RWAOracleRateCheck, the change is rounded only once, and rounding up always tightens the constraints.

Update from Client: PR to round up for RWAOracleRateCheck: a998c11559457b4d789cfda75903ee489f827cb9.

6.2 [Low] Changing oracle type does not remove previous oracle data

File(s): contracts/lending/fluxOracles/FluxOracle.sol

Description: The FluxOracle contract allows for the oracle type to be changed for each asset. However, when the oracle type is changed, the data previously stored in the fTokenToHardcodedPrice, fTokenToRWAOracle, and fTokenToChainlinkOracle mappings for the given asset is not removed. This can result in unintended behavior. It is good practice to remove unused data to prevent such issues and to save on gas costs. Below we present setFTokenToOracleType(...) function responsible for changing the oracle type:

```
function setFTokenToOracleType(
  address fToken,
  OracleType oracleType
) external override onlyRole(DEFAULT_ADMIN_ROLE) {
  fTokenToOracleType[fToken] = oracleType;
  ////////////////////////////////////////////////////////////////////////////////
  // @audit Data from mappings fTokenToHardcodedPrice, fTokenToRWAOracle and  //
  //              fTokenToChainlinkOracle is not removed.                       //
  ////////////////////////////////////////////////////////////////////////////////
  emit FTokenToOracleTypeSet(fToken, oracleType);
}
```

Recommendation(s): Consider removing any unused data when changing the oracle type for a given asset.

Status: Fixed

Update from the client: Cleared stale data in this PR: c39b3532de80600b201d801f28a699ae93bd07de.

6.3 [Low] Lack of a two-step process for transferring ownership

File(s): contracts/lending/fTokenOracle/fTokenOracle.sol

Description: The fTokenOracle contract inherits OpenZeppelin's Ownable contract, which enables ownership to be transferred in a single step. However, this one-step transfer process can be prone to errors and may cause significant damage to the protocol if a mistake occurs. The function is presented below:

```
/////////////////////////////////////////////////////////////////////////////////////////
// @audit Providing an incorrect address will result in the immediate loss of ownership
/////////////////////////////////////////////////////////////////////////////////////////
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}
```

Recommendation(s): We recommend implementing a two-step ownership transfer process, such as the propose-accept scheme. You can refer to the Ownable2Step.sol contract from OpenZeppelin for an example.

Status: Fixed

Update from the client: Added Ownable2Step in this PR: 0b86a1b554a8c345faef8b22045c718e52e2d533.

6.4 [Low] Use AccessControlDefaultAdminRules

File(s): contracts/lending/rwaOracles/RWAOracleExternalComparisonCheck.sol, contracts/lending/rwaOracles/RWAOracleRateCheck.sol, contracts/lending/fluxOracles/FluxOracle.sol

Description: The FluxOracle, RWAOracleRateCheck, and RWAOracleExternalComparisonCheck contracts inherit from AccessControl. However, the AccessControl contract allows for potentially insecure behavior with the DEFAULT_ADMIN_ROLE, as multiple addresses can be granted this role and it is its own admin, meaning it has permission to grant and revoke the default admin role. A more secure solution is the AccessControlDefaultAdminRules contract, which implements additional risk mitigations on top of AccessControl:

- only one account holds the DEFAULT_ADMIN_ROLE since deployment until it's potentially renounced;
- enforces a 2-step process to transfer the DEFAULT_ADMIN_ROLE to another account;
- enforces a configurable delay between the two steps, with the ability to cancel before the transfer is accepted;
- the delay can be changed by scheduling;
- it is not possible to use another role to manage the DEFAULT_ADMIN_ROLE;

Recommendation(s): Consider using AccessControlDefaultAdminRules to increase the security of role management.

Status: Acknowledged

Update from the client: Understand the benefits of AccessControlDefaultAdminRules; however, we don't believe we need the added features/complexity at this point in time.

6.5 [Low] `_setFTokenToChainlinkOracle(...)` does not check oracle decimals

File(s): contracts/lending/fluxOracles/FluxOracle.sol

Description: The _setFTokenToChainlinkOracle(...) function assigns a specified Chainlink oracle to an fToken. However, there is no validation process in place to ensure that the number of decimals returned by the Chainlink oracle falls within an acceptable range. The function is presented below:

```
function _setFTokenToChainlinkOracle(address fToken, address chainlinkOracle, uint256 maxChainlinkOracleTimeDelay )
 ↪ private {
  if (fTokenToOracleType[fToken] != OracleType.CHAINLINK) {
    revert InvalidOracleType(
      OracleType.CHAINLINK,
      fTokenToOracleType[fToken]
    );
  }
  address underlying = ICTokenLike(fToken).underlying();
  fTokenToChainlinkOracle[fToken].scaleFactor = (10 ** (36 - uint256(IERC20Like(underlying).decimals()) -
      ////////////////////////////////////////////////////////////////////////////////////
      // @audit The decimals from Oracle are not checked
      ////////////////////////////////////////////////////////////////////////////////////
      uint256(AggregatorV3Interface(chainlinkOracle).decimals())));
  fTokenToChainlinkOracle[fToken].oracle = AggregatorV3Interface(chainlinkOracle);
  fTokenToChainlinkOracle[fToken].maxChainlinkOracleTimeDelay = maxChainlinkOracleTimeDelay;
}
```

Recommendation(s): Verify the number of decimals returned by the Chainlink oracle.

Status: Acknowledged

Update from the client: The contract works as intended in cases where `underlyingDecimals + chainlinkDecimals <= 36`. If the sum is > 36, the function will underflow. If the sum is == 36, the scaleFactor will be 1. If the sum is < 36, the scaleFactor will be some value >= 1e2. Check is not needed.

Update from Nethermind: We believe that a check for decimals should be performed to ensure that the Chainlink response is not corrupted. This check would be consistent with the other checks for Chainlink response decimals in the `fTokenOracle` and `RWAOracleExternalComparisonCheck` contracts.

6.6 [Info] Events in RWA oracles differ

File(s): `contracts/lending/rwaOracles/RWAOracleRateCheck.sol`, `contracts/lending/rwaOracles/RWAOracleExternalComparisonCheck.sol`

Description: The `RWAOracleRateCheck` and `RWAOracleExternalComparisonCheck` contracts can both be assigned as oracles to an `fToken` with the same specified in the `FluxOracle`. As such, they are interchangeable. However, it should be noted that the events emitted when the price is set contain a varying number of fields. This discrepancy may pose a problem if these logs are utilized by off-chain infrastructure. The events in both contracts are presented below:

```
//////////////////////////////////////////////////////////////////////////////////
// @audit RWAOracleRateCheck
//////////////////////////////////////////////////////////////////////////////////
event RWAPriceSet(
  int256 oldPrice,
  int256 newPrice,
  uint256 timestamp
);
```

```
//////////////////////////////////////////////////////////////////////////////////
// @audit RWAOracleExternalComparisonCheck
//////////////////////////////////////////////////////////////////////////////////
event RWAPriceSet(
  int256 oldChainlinkPrice,
  uint80 indexed oldRoundId,
  int256 newChainlinkPrice,
  uint80 indexed newRoundId,
  int256 oldRWAPrice,
  int256 newRWAPrice
);
```

Recommendation(s): Maintain consistency in the number of fields included in the RWAPriceSet event. Alternatively, change the event names to correspond to the contracts that emit them.

Status: Fixed

Update from the client: We have changed the name of the event in `RWAOracleExternalComparisonCheck` in this commit. 22716a82e7382ab2479d6fe74f79c

6.7 [Info] Inconsistent behavior on `price == 0`

File(s): `contracts/lending/fluxOracles/FluxOracle.sol`

Description: The `FluxOracle` contract supports three types of oracles: hardcoded, RWA, and Chainlink. However, these oracle types do not exhibit consistent behavior when the price is equal to `0`. Here is a summary of the behavior for each oracle type:

- Hardcoded: The price can be set to and retrieved as `0`;
- RWA: The oracle cannot be set if the returned price is `0`. The price can be retrieved as `0`;
- Chainlink: The price can be retrieved as `0` (but reverts if less than `0`);

This inconsistent behavior with zero values may lead to incorrect assumptions when using the oracles. While checking for a zero value when setting the RWA oracle is good practice, it creates an assumption that this oracle will always work correctly and does not check the returned price on a get call. Additionally, the price set for the hardcoded oracle is not checked and can be `0`. The Chainlink price is only considered invalid if it is less than `0`, allowing for `0` to be a valid price.

Recommendation(s): Consider establishing a unified definition of a zero price for all types of oracles and implementing the set and get functions to behave consistently when the price is zero.

Status: Fixed

Update from the client: FluxOracle contract now prevents setting 0 price for Hardcoded type and retrieving zero prices for all oracle types. Fixed in: 48d5c4052795b082e876926f18f884927bb653b9.

6.8 [Info] Incorrect check for decimals

File(s): contracts/lending/rwaOracles/RWAOracleExternalComparisonCheck.sol

Description: The constructor includes a check to ensure that the Chainlink oracle returns at least 4 decimals. The relevant code is shown below:

```
constructor(
    int256 initialPrice,
    address _chainlinkOracle,
    string memory _description,
    address admin,
    address _setter
  ) {
    chainlinkOracle = AggregatorV3Interface(_chainlinkOracle);
    if (chainlinkOracle.decimals() < 4) revert CorruptedChainlinkResponse();
    ....
}
```

However, it should be noted that the current minimum number of decimals returned by Chainlink is 8 for non-ETH pairs. In the event that the returned decimals are less than 8 but greater than 3, which could indicate a malfunction of Chainlink, this would not be detected by the contract.

Recommendation(s): Consider checking if the decimals returned by Chainlink is 8.

Status: Fixed

Update from the client: We have changed the check to `decimals==8` in this commit. f983d5487e568016ad1ae2e4db72c2ec50d2b74d.

6.9 [Info] No check for fToken validity

File(s): contracts/lending/fluxOracles/FluxOracle.sol

Description: In the `FluxOracle` contract, the oracle type can be assigned to an `fToken` through the `fTokenToOracleType` mapping. However, there is no check in place to ensure that the `fToken` is a valid address. This could be implemented in a similar manner to the `fTokenOracle` contract:

```
if (!IFToken(_fToken).isCToken()) {
  revert InvalidFToken();
}
```

Recommendation(s): Consider checking the validity of the `fToken` when assigning the oracle type to the asset.

Status: Fixed

Update from the client: Check added in this PR: f19e5a38b78153ea436632f480b73194cdcda930.

6.10 [Best Practice] Use of magic numbers

File(s): contracts/lending/fTokenOracle/fTokenOracle.sol

Description: The `fTokenOracle` contract calculates the `scaleFactor` using the following equation:

```
////////////////////////////////////////////////////////////////////////////////////
// @audit Number 8 is not explained
////////////////////////////////////////////////////////////////////////////////////
scaleFactor = 10 ** (priceFeedDecimals - 8 + fTokenUnderlyingDecimals);
```

The constant `8` is used in the calculation, but its purpose is not explained.

Recommendation(s): It is recommended to store constants as variables and provide a description that explains their purpose. This would improve the readability and maintainability of the code.

Status: Fixed

Update from the client: Removed magic number in this PR: 0b86a1b554a8c345faef8b22045c718e52e2d533.

6.11 [Best Practice] `renounceOwnership(...)` function should be disabled

File(s): `contracts/lending/fTokenOracle/fTokenOracle.sol`

Description: The `fTokenOracle` contract, which is inherited from the `Ownable` contract, includes the `renounceOwnership(...)` function. This function allows the owner to be removed from the contract. Below is a presentation of the function:

```
//////////////////////////////////////////////////////////////////////////////////
// @audit Renouncing ownership will leave the contract without an owner
//////////////////////////////////////////////////////////////////////////////////
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Recommendation(s): If the `fTokenOracle` is intended to operate with an owner, consider disabling the `renounceOwnership(...)` function.

Status: Acknowledged

Update from the client:

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document that describes the design and technical details of the smart contract. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

The `Ondo Finance` team has provided extensive documentation in the Markdown format within the audit repository. This documentation encompasses a general overview of the protocol and detailed insights into each component. Additional documents about the mathematical logic behind certain aspects of the project were also shared. Moreover, the team conducted a comprehensive code walkthrough and maintained open communication to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

The Ondo team has placed a significant focus on testing their code to ensure its quality. They have given particular attention to two essential testing methods: unit testing and fuzzing testing. For unit testing, they use the Forge testing framework, which helps them to thoroughly test individual components of their code in a controlled environment. Additionally, they use the Echidna tool for fuzzing testing. This involves generating random inputs to identify potential issues and edge cases. By employing both of these testing methods, the Ondo team can confidently guarantee that their code is reliable, resilient, and capable of handling a wide range of scenarios.

8.1 Contracts Compilation Output

```
> forge build
[] Compiling...
[] Compiling 28 files with 0.5.17
[] Compiling 221 files with 0.8.16
[] Compiling 3 files with 0.6.12
[] Compiling 5 files with 0.8.7
[] Solc 0.6.12 finished in 46.28ms
[] Solc 0.8.7 finished in 1.02s
[] Solc 0.5.17 finished in 2.07s
[] Solc 0.8.16 finished in 73.49s
Compiler run successful (with warnings)
```

8.2 Tests Output

```
> yarn test-forge-oracles

Running 24 tests for RWAOracleExternalComparisonCheck.t.sol:RWAOracleExternalComparisonCheckTest
[PASS] test_absoluteDeviationAtLowerBound() (gas: 93031)
[PASS] test_absoluteDeviationAtUpperBound() (gas: 93374)
[PASS] test_absoluteDeviationOutsideLowerBound() (gas: 58557)
[PASS] test_absoluteDeviationOutsideUpperBound() (gas: 58544)
[PASS] test_bypassCheckCLLowerBound() (gas: 93367)
[PASS] test_bypassCheckCLLowerBoundFailure() (gas: 60407)
[PASS] test_bypassCheckCLTooHigh() (gas: 87472)
[PASS] test_bypassCheckCLTooLow() (gas: 89006)
[PASS] test_bypassCheckCLUpperBound() (gas: 93728)
[PASS] test_bypassCheckCLUpperBoundFailure() (gas: 60297)
[PASS] test_constructorFailNegativePrice() (gas: 73041)
[PASS] test_constructorFailRoundTooOld() (gas: 79151)
[PASS] test_constrainedSetter_maxDeviationPositive() (gas: 25157)
[PASS] test_constrainedSetter_setPrice() (gas: 28056)
[PASS] test_constrainedSetter_setPrice_fail_accessControl() (gas: 75359)
[PASS] test_constrainedSetter_setPrice_fail_negative() (gas: 10980)
[PASS] test_constrainedSetter_setPrice_fail_tooLarge() (gas: 17111)
[PASS] test_constrainedSetter_setPrice_fail_tooSmall() (gas: 17275)
[PASS] test_constrainedSetter_setPrice_fail_tooSoon() (gas: 13070)
[PASS] test_constrainedSetter_setPrice_fail_zeroPrice() (gas: 10912)
[PASS] test_fuzz_multipleUpdates(uint256) (runs: 256, : 746114, ~: 747398)
[PASS] test_getPriceData() (gas: 29210)
Test result: ok. 14 passed; 0 failed; finished in 2.20s

Running 17 tests for forge-tests/lending/oracle/fTokenOracle/fUSDC-USD.t.sol:Test_fUSDC_USD_Oracle_ETH
[PASS] test_fTokenOracle_description() (gas: 9816)
[PASS] test_fTokenOracle_fail_unsupportedPriceFeed() (gas: 403516)
[PASS] test_fTokenOracle_getLatestPrice() (gas: 45048)
[PASS] test_fTokenOracle_getLatestPrice_fail_negativePrice() (gas: 1030715)
[PASS] test_fTokenOracle_getLatestPrice_fail_staleRound() (gas: 1010627)
[PASS] test_fTokenOracle_getLatestPrice_fail_staleUpdateTime() (gas: 1031562)
[PASS] test_fTokenOracle_getLatestPrice_fail_zeroPrice() (gas: 1010813)
[PASS] test_fTokenOracle_getPrice() (gas: 44983)
[PASS] test_fTokenOracle_getPrice_fail_negativePrice() (gas: 1030438)
[PASS] test_fTokenOracle_getPrice_fail_zeroPrice() (gas: 1010623)
[PASS] test_fTokenOracle_getPrice_increase() (gas: 949811)
[PASS] test_fTokenOracle_has_owner() (gas: 7707)
```

```
[PASS] test_fTokenOracle_scaleFactor() (gas: 5537)
[PASS] test_fTokenOracle_setMaxOracleTimeDelay() (gas: 19106)
[PASS] test_fTokenOracle_setMaxOracleTimeDelay_fail_accessControl() (gas: 11404)
[PASS] test_fTokenOracle_transferOwnership() (gas: 13470)
[PASS] test_fTokenOracle_transferOwnership_fail_accessControl() (gas: 11496)
Test result: ok. 17 passed; 0 failed; finished in 16.74s

Running 17 tests for forge-tests/lending/oracle/fTokenOracle/fUSDC-ETH.t.sol:Test_fUSDC_ETH_Oracle_ETH
[PASS] test_fTokenOracle_description() (gas: 9816)
[PASS] test_fTokenOracle_fail_unsupportedPriceFeed() (gas: 403516)
[PASS] test_fTokenOracle_getLatestPrice() (gas: 48546)
[PASS] test_fTokenOracle_getLatestPrice_fail_negativePrice() (gas: 1030715)
[PASS] test_fTokenOracle_getLatestPrice_fail_staleRound() (gas: 1010627)
[PASS] test_fTokenOracle_getLatestPrice_fail_staleUpdateTime() (gas: 1031562)
[PASS] test_fTokenOracle_getLatestPrice_fail_zeroPrice() (gas: 1010813)
[PASS] test_fTokenOracle_getPrice() (gas: 46239)
[PASS] test_fTokenOracle_getPrice_fail_negativePrice() (gas: 1030438)
[PASS] test_fTokenOracle_getPrice_fail_zeroPrice() (gas: 1010623)
[PASS] test_fTokenOracle_getPrice_increase() (gas: 949811)
[PASS] test_fTokenOracle_has_owner() (gas: 7707)
[PASS] test_fTokenOracle_scaleFactor() (gas: 5537)
[PASS] test_fTokenOracle_setMaxOracleTimeDelay() (gas: 19106)
[PASS] test_fTokenOracle_setMaxOracleTimeDelay_fail_accessControl() (gas: 11404)
[PASS] test_fTokenOracle_transferOwnership() (gas: 13470)
[PASS] test_fTokenOracle_transferOwnership_fail_accessControl() (gas: 11496)
Test result: ok. 17 passed; 0 failed; finished in 16.74s

Running 26 tests for forge-tests/.../Test_Flux_Oracle_RWAOracleExternalComparisonCheck
[PASS] test_fluxOracle_accessControl() (gas: 36674)
[PASS] test_fluxOracle_getChainlinkOraclePrice_18Decimals() (gas: 491757)
[PASS] test_fluxOracle_getChainlinkOraclePrice_6Decimals() (gas: 503296)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_negativePrice() (gas: 494219)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_roundId() (gas: 474372)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_staleUpdate() (gas: 494499)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_updatedAt() (gas: 494272)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_zeroRoundId() (gas: 474317)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_zeroUpdatedAt() (gas: 454385)
[PASS] test_fluxOracle_getTokenizedRWAPrice_updatePrice() (gas: 148819)
[PASS] test_fluxOracle_setFTokenToChainLinkOracle_fail_invalidType() (gas: 26291)
[PASS] test_fluxOracle_setFTokenToChainlinkOracle() (gas: 128735)
[PASS] test_fluxOracle_setFTokenToChainlinkOracle_fail_accessControl() (gas: 79944)
[PASS] test_fluxOracle_setFTokenToOracleType_chainlink() (gas: 22938)
[PASS] test_fluxOracle_setFTokenToOracleType_fail_accessControl() (gas: 43072)
[PASS] test_fluxOracle_setFTokenToOracleType_hardcoded() (gas: 20095)
[PASS] test_fluxOracle_setFTokenToOracleType_tokenizedRWA() (gas: 20092)
[PASS] test_fluxOracle_setFTokenToRWAOracle() (gas: 45197)
[PASS] test_fluxOracle_setFTokenToRWAOracle_fail_accessControl() (gas: 79691)
[PASS] test_fluxOracle_setFTokenToRWAOracle_fail_invalidType() (gas: 26781)
[PASS] test_fluxOracle_setFTokenToRWAOracle_fail_notRWAOracle() (gas: 21131)
[PASS] test_fluxOracle_setFTokenToRWAOracle_invalidRWAOraclePrice() (gas: 26908)
[PASS] test_fluxOracle_setHardcodedPrice() (gas: 32142)
[PASS] test_fluxOracle_setHardcodedPrice_fail_accessControl() (gas: 77612)
[PASS] test_fluxOracle_setHardcodedPrice_fail_invalidType() (gas: 24628)
[PASS] test_getUnderlying_fail_invalidType() (gas: 16445)
Test result: ok. 26 passed; 0 failed; finished in 16.74s

Running 26 tests for forge-tests/.../Test_Flux_Oracle_Percent_Constrained_RWA_Oracle
[PASS] test_fluxOracle_accessControl() (gas: 36696)
[PASS] test_fluxOracle_getChainlinkOraclePrice_18Decimals() (gas: 491745)
[PASS] test_fluxOracle_getChainlinkOraclePrice_6Decimals() (gas: 503284)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_negativePrice() (gas: 494207)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_roundId() (gas: 474338)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_staleUpdate() (gas: 494487)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_updatedAt() (gas: 494260)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_zeroRoundId() (gas: 474305)
[PASS] test_fluxOracle_getChainlinkOraclePrice_fail_zeroUpdatedAt() (gas: 454373)
[PASS] test_fluxOracle_getTokenizedRWAPrice_updatePrice() (gas: 83984)
[PASS] test_fluxOracle_setFTokenToChainLinkOracle_fail_invalidType() (gas: 26336)
```

```
[PASS] test_fluxOracle_setFTokenToChainlinkOracle() (gas: 128735)
[PASS] test_fluxOracle_setFTokenToChainlinkOracle_fail_accessControl() (gas: 79922)
[PASS] test_fluxOracle_setFTokenToOracleType_chainlink() (gas: 22938)
[PASS] test_fluxOracle_setFTokenToOracleType_fail_accessControl() (gas: 43072)
[PASS] test_fluxOracle_setFTokenToOracleType_hardcoded() (gas: 20139)
[PASS] test_fluxOracle_setFTokenToOracleType_tokenizedRWA() (gas: 20092)
[PASS] test_fluxOracle_setFTokenToRWAOracle() (gas: 45087)
[PASS] test_fluxOracle_setFTokenToRWAOracle_fail_accessControl() (gas: 79691)
[PASS] test_fluxOracle_setFTokenToRWAOracle_fail_invalidType() (gas: 26758)
[PASS] test_fluxOracle_setFTokenToRWAOracle_fail_notRWAOracle() (gas: 21108)
[PASS] test_fluxOracle_setFTokenToRWAOracle_invalidRWAOraclePrice() (gas: 26930)
[PASS] test_fluxOracle_setHardcodedPrice() (gas: 32164)
[PASS] test_fluxOracle_setHardcodedPrice_fail_accessControl() (gas: 77634)
[PASS] test_fluxOracle_setHardcodedPrice_fail_invalidType() (gas: 24628)
[PASS] test_getUnderlying_fail_invalidType() (gas: 16445)
Test result: ok. 26 passed; 0 failed; finished in 16.74s

Running 1 test for forge-tests/.../SHVOracleIntegration.t.sol:Test_SHV_Oracle_Integration
[PASS] test_RWAOraclePriceUpdates() (gas: 357290)
Test result: ok. 1 passed; 0 failed; finished in 25.12s
```

8.3 Code Coverage

Unable to run code coverage.

8.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

Disclaimer