Security Review Report
NM-0703 MELLOW - ORACLE SUBMITTER SECURITY REVIEW

(November 17, 2025)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 3

4 System Overview 4
4.1 Standard Price Report Submission . . . 4
4.2 Suspicious Report Acceptance . . . 4
4.3 Chainlink-Compatible Price Feed . . . 4

5 Risk Rating Methodology 5

6 Issues 6
6.1 [Info] Calculation of `latestAnswer` introduces minor precision loss due to double rounding . . . 6
6.2 [Info] The `latestRoundData()` function returns misleading timestamps . . . 7

7 Documentation Evaluation 8

8 Test Suite Evaluation 9

9 About Nethermind 10

1 Executive Summary

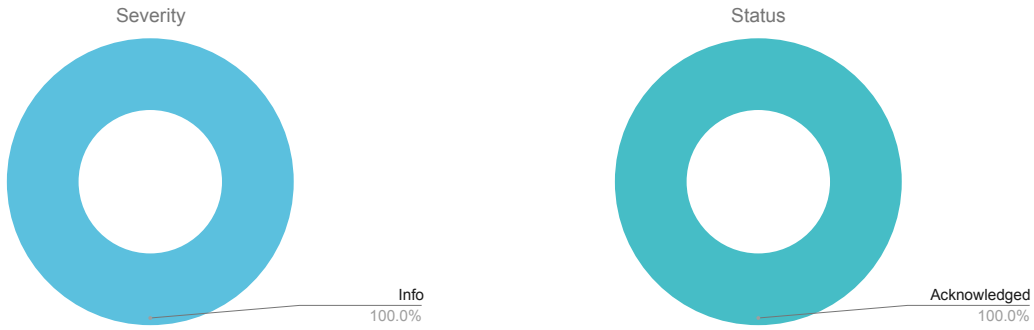
This document presents the security review performed by Nethermind Security for Mellow Protocol smart contracts. The scope of this audit was specifically focused on the newly developed `OracleSubmitter` contract. The `OracleSubmitter` contract is a specialized adapter designed to serve as an on-chain, Chainlink-compatible price feed for the Mellow Protocol. Its primary function is to act as an interface by submitting price reports to the main `Oracle` contract (which validates them), reading back the validated report (including its suspicion status), caching it, and exposing the latest non-suspicious price for the system's `baseAsset`.

To enforce this architecture, the `OracleSubmitter` contract's address is designed to be the sole holder of `SUBMIT_REPORTS_ROLE` and `ACCEPT_REPORT_ROLE` in the main protocol's `vault`.

The audited code comprises 76 lines of code written in the Solidity language. The audit included the `OracleSubmitter` contract.

The audit was performed using (a) manual analysis of the codebase and (b) automated analysis tools. **Along this document, we report** 2 points of attention, both are classified as `Informational` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. 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.



(a) (b)

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

Summary of the Audit

Audit Type	Security Review
Initial Report	November 17, 2025
Final Report	November 17, 2025
Initial Commit	d3bf393
Final Commit	d3bf393
Documentation Assessment	Low
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/oracles/OracleSubmitter.sol	76	4	5.3%	18	98
	Total	**76**	**4**	**5.3%**	**18**	**98**

3 Summary of Issues

	Finding	Severity	Update
1	Calculation of `latestAnswer` introduces minor precision loss due to double rounding	Info	Acknowledged
2	The `latestRoundData()` function returns misleading timestamps	Info	Acknowledged

4 System Overview

For a comprehensive understanding of the entire Mellow Protocol system, please refer to the official audit report: NM0587-FINAL_Mellow. This section focuses exclusively on the newly added `OracleSubmitter` contract.

The `OracleSubmitter` contract is a specialized adapter designed to serve as an on-chain, Chainlink-compatible price feed for the Mellow Protocol. Its primary function is to act as a complete interface by submitting price reports to the main `IOracle` contract (which validates them), reading back the validated report—including its suspicion status—caching it, and exposing the latest non-suspicious price of the system's `baseAsset`.

To enforce this architecture, the `OracleSubmitter` contract's address is designed to be the sole holder of the `SUBMIT_REPORTS_ROLE` and `ACCEPT_REPORT_ROLE` on the main protocol's `vault`. This design ensures it acts as a mandatory proxy for all price submissions and acceptances. This allows external protocols (e.g., Gearbox) to consume Mellow's `baseAsset` price using the standard `latestRoundData()` interface.

4.1 Standard Price Report Submission

The primary flow for updating the price feed begins when an authorized `Submitter` (an address granted the `SUBMIT_REPORTS_ROLE` on the `OracleSubmitter` contract) calls `submitReports(...)`.

- **Submission**: The authorized `Submitter` calls `submitReports(...)` with an array of reports. The protocol enforces that the first report in the array must be for the `baseAsset`.
- **Forwarding**: The `OracleSubmitter` contract (acting as the sole role-holder) then forwards the raw reports to the main `IOracle` contract by calling `oracle.submitReports(...)`. The main `IOracle` contains the logic to validate the price, check for deviations, and flag a report as suspicious.
- **Caching**: After submission, the `OracleSubmitter` immediately queries the main `IOracle` using `oracle.getReport(...)` to retrieve the detailed, validated report (which includes the `isSuspicious` flag). This detailed report is then cached locally in the `_reports` mapping.
- **Price Update**: If the new report is for the `baseAsset` and is not flagged as suspicious (`!report.isSuspicious`), the `OracleSubmitter` contract updates its internal `latestAnswer` by calling `_updateLatestAnswer(...)`. If a `baseAsset` report is suspicious, it is cached, but `latestAnswer` is not updated.

4.2 Suspicious Report Acceptance

If the main `IOracle` flags a `baseAsset` report as suspicious (e.g., due to significant price deviation), the `latestAnswer` is not updated, and the price feed remains stale. This situation requires manual intervention from an authorized `Accepter` (an address granted the `ACCEPT_REPORT_ROLE` on the `OracleSubmitter` contract).

- **Manual Acceptance**: This authorized actor calls the `acceptReports(...)` function, providing the asset, price, and timestamp of the suspicious report.
- **Forwarding Acceptance**: This call is proxied by the `OracleSubmitter` contract (acting as the sole role-holder) to the main `IOracle` via `oracle.acceptReport(...)`, which clears the `isSuspicious` flag in the main oracle.
- **Price Update**: When `acceptReports(...)` is called, the `OracleSubmitter` checks if the accepted report is for the `baseAsset`. If it is, the contract immediately updates its `latestAnswer` with the now-accepted price. This ensures the feed becomes "unstuck" and reflects the manually verified price.
- **Tracking**: The contract records the timestamp of this manual acceptance in the `acceptedAt` mapping.

4.3 Chainlink-Compatible Price Feed

The primary purpose of the `OracleSubmitter` is to be consumed by other protocols. It achieves this by exposing a standard Chainlink `AggregatorV3Interface`.

- **`latestRoundData()`**: This function provides the standard interface for price feed consumption. It returns the `latestAnswer` (the price) and `updatedAt` (the timestamp of the last update). As Mellow's oracle does not use round IDs, `roundId` and `answeredInRound` are hardcoded to `0`.
- **Price Calculation**: The `latestAnswer` is not stored as a simple price. The main `IOracle` provides `priceD18`, which represents *shares per asset*, scaled by 1e18. To be useful as a price feed, this value must be inverted to *assets per share*. The `_updateLatestAnswer(...)` function calculates this as `int256(1e36 / uint256(priceD18))`.
- **Example**: If the `baseAsset` is USDC (6 decimals) and the `IOracle` reports a `priceD18` of 1e30 (1e18 shares for 1e-12 USDC), the `latestAnswer` becomes `1e36 / 1e30 = 1e6`. This `1e6` correctly represents the price of one share in terms of the asset's smallest unit (1 USDC = 1e6).

5 Risk Rating Methodology

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

Likelihood measures 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 an attacker exploits the finding. 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 Security 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 pass to the client formally;

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 [Info] Calculation of `latestAnswer` introduces minor precision loss due to double rounding

File(s): src/oracles/OracleSubmitter.sol

Description: The `OracleSubmitter` contract is designed to function as a Chainlink-compatible price feed adapter. It exposes functions like `latestRoundData(...)` which return the `latestAnswer` state variable, representing the price of the base asset per share (assets/shares).

This `latestAnswer` is calculated and set in the `_updateLatestAnswer(...)` function by inverting the `priceD18` value (which represents shares/assets). The issue is that this process involves a double-rounding error.

1. The incoming `priceD18` value is typically calculated externally (e.g., in `OracleHelper`) using `Math.mulDiv(...)`. This division calculates `(shares * 1e18) / assets`, rounding the result _down_ (floor);
2. The `_updateLatestAnswer(...)` function then takes this already rounded-down `priceD18` and uses it as a denominator in another division: `1e36 / priceD18`;

```
1  function _updateLatestAnswer(uint224 priceD18) internal {
2   // @audit The priceD18 value is the result of a previous division that rounded down.
3   // @audit-issue Using this rounded-down value as a denominator in a new division
4   // causes the final result (latestAnswer) to be rounded up, amplifying the inaccuracy.
5   latestAnswer = int256(1e36 / uint256(priceD18));
6   updatedAt = uint32(block.timestamp);
7  }
```

Because the denominator (`priceD18`) is slightly smaller than its true value (due to the first rounding-down), the final `latestAnswer` (which also rounds down) will be slightly _larger_ than the true asset/share price. This introduces a minor but persistent upward bias in the reported price. While this value is understood to be an _effective_ price, reflecting vault state (including continuous fees and queued actions), this specific rounding behavior is an artifact of the calculation method.

Recommendation(s): Consider documenting this calculation behavior. Since the protocol's price already represents an effective rate rather than a "spot" price, this minor precision artifact may be acceptable. However, any integrator relying on this oracle as a Chainlink-compatible feed should be made aware that the `latestAnswer` is subject to a small, persistent upward bias due to this double-rounding.

Status: Acknowledged

6.2 [Info] The `latestRoundData()` function returns misleading timestamps

File(s): src/oracles/OracleSubmitter.sol

Description: The `OracleSubmitter` contract provides a Chainlink-compatible `latestRoundData()` interface. This function is expected to return five values, including an `answer`, a `startedAt` timestamp (when the price round began), and an `updatedAt` timestamp (when the answer was last updated).

The issue is that the `latestRoundData()` function implementation uses the contract's internal `updatedAt` state variable for _both_ the `startedAt` and `updatedAt` return values.

```
// src/oracles/OracleSubmitter.sol
function latestRoundData() public view returns (uint80, int256, uint256, uint256, uint80) {
    uint32 timestamp = updatedAt;
// @audit-issue The same `updatedAt` value is returned for both `startedAt` and `updatedAt`.
    return (0, latestAnswer, timestamp, timestamp, 0);
}
```

This internal `updatedAt` variable is set in the `_updateLatestAnswer(...)` helper function. This helper is called in two scenarios:

1. In `submitReports(...)`, _only if_ the submitted report is **not** suspicious;
2. In `acceptReports(...)`, when a (previously suspicious) report is manually accepted;

```
// src/oracles/OracleSubmitter.sol
function _updateLatestAnswer(uint224 priceD18) internal {
// ...
// @audit `updatedAt` is set to the current timestamp of finalization.
    updatedAt = uint32(block.timestamp);
}

function submitReports(...) external onlyRole(oracle.SUBMIT_REPORTS_ROLE()) {
// ...
// @audit This is only called if the report is *not* suspicious.
      if (!report.isSuspicious && asset == baseAsset) {
        _updateLatestAnswer(report.priceD18);
      }
// ...
}
```

The problem arises when a report for the `baseAsset` is submitted and flagged as suspicious. The `submitReports(...)` function will _not_ update `updatedAt`. Only later, when `acceptReports(...)` is called, will `updatedAt` be set to the _acceptance_ timestamp.

This means that for a suspicious report, `latestRoundData()` will return the _acceptance time_ as both the `startedAt` and `updatedAt` time. This is misleading for consumers of the oracle, as it hides the true submission time (when the "round" effectively started) and the latency between submission and acceptance.

Recommendation(s): Consider revisiting the logic for how timestamps are stored and returned in `latestRoundData()`. A more accurate implementation might distinguish between the initial submission time of a price report (which could be considered `startedAt`) and the time it was finalized and accepted (the current `updatedAt`).

Alternatively, if this behavior is intended, consider adding clear documentation to the `latestRoundData()` function explaining that both `startedAt` and `updatedAt` will reflect the finalization timestamp of the latest valid price, especially in cases where a suspicious report is manually accepted.

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 is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. 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 describing the smart contract's design and technical details. 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.

Remarks about the `OracleSubmitter` documentation

No documentation for the `OracleSubmitter` contract was provided, but Mellow's team addressed all the questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

No tests were provided for the `OracleSubmitter` contract specifically. The current test suite focuses on the `Oracle` and `OracleHelper` contracts. The Nethermind Security team recommends creating a comprehensive test suite for the `OracleSubmitter` contract as well.

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.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.