// Security Assessment

02.19.2025 - 02.21.2025

Fetch Oracle - Fetch Oracle Contracts Updates

Fetch Oracle

HALBORN

Fetch Oracle - Fetch Oracle Contracts Updates

Prepared by: HALBORN

Last Updated 03/07/2025

Date of Engagement by: February 19th, 2025 - February 21st, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	0	0	0	8

TABLE OF CONTENTS

1. Introduction

Fetch Oracle engaged Halborn to conduct a security assessment on their smart contracts beginning on February 19th, 2025 and ending on February 21st, 2025. The security assessment was scoped to the smart contracts provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

The Fetch Oracle codebase in scope consists of updates made to a set of smart contracts designed to act as a decentralized oracle protocol.

2. Assessment Summary

Halborn was provided 3 days for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly acknowledged by the Fetch Oracle team. The main ones are the following:

- Implement consistent price validation across all query types.
- Update NATSPEC comments in the the removeManagedQuery() function or implement a deprecation system to remove managed queries.
- Move the __reserved array declaration to the end of all storage variable declarations.

3. Test Approach And Methodology

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture, purpose and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could led to arithmetic related vulnerabilities.
- Local testing.
- Static analysis of security for scoped contract, and imported functions (`Slither`).

4. Caveats

The current security review was focused solely on evaluating the changes introduced between a previous assessment from `Halborn` and the current state of the files (a `diff` assessment). While this review aimed to provide a comprehensive evaluation of the protocol's security posture, it is important to consider this limitations when interpreting the findings and recommendations.

5. Static Analysis Report

5.1 Description

`Halborn` used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After `Halborn` verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

5.2 Output

There were no findings that matched the scope of this assessment.

6. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

6.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

6.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

6.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

7. SCOPE

FILES AND REPOSITORY

(a) Repository: fetch-contracts

(b) Assessed Commit ID: 3fbd55f

(c) Items in scope:

- contracts/Autopay.sol
- contracts/FetchFlex.sol
- contracts/FetchToken.sol
- contracts/Governance.sol
- contracts/QueryDataStorage.sol
- contracts/usingfetch/UsingFetch.sol
- contracts/usingfetch/UsingFetchUpgradeReady.sol
- contracts/interfaces/IERC20.sol
- contracts/interfaces/IFetch.sol
- contracts/interfaces/IFetchFlex.sol
- contracts/interfaces/IFetchToken.sol
- contracts/interfaces/IOracle.sol
- contracts/interfaces/IERC2362.sol
- contracts/interfaces/IMappingContract.sol
- contracts/interfaces/IQueryDataStorage.sol

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

8. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
IMPROPER PLACEMENT OF RESERVED ARRAY	INFORMATIONAL	ACKNOWLEDGED - 02/27/2025
MANAGEDQUERY BECOMES PERMANENT AFTER FIRST VALUE SUBMISSION	INFORMATIONAL	ACKNOWLEDGED - 02/27/2025
MISSING ERROR DESCRIPTIONS	INFORMATIONAL	FUTURE RELEASE - 02/27/2025
INCORRECT NATSPEC DOCUMENTATION	INFORMATIONAL	FUTURE RELEASE - 02/27/2025
MISSING EVENTS	INFORMATIONAL	FUTURE RELEASE - 02/27/2025
CONSIDER USING FLEXIBLE PRAGMA FOR CONTRACT INTERFACES	INFORMATIONAL	FUTURE RELEASE - 02/27/2025
UNUSED FUNCTION	INFORMATIONAL	FUTURE RELEASE - 02/27/2025
MODIFIED OPENZEPPELIN CONTRACTS	INFORMATIONAL	FUTURE RELEASE - 02/27/2025

9. FINDINGS & TECH DETAILS

9.1 IMPROPER PLACEMENT OF RESERVED ARRAY

// INFORMATIONAL

Description

In the FetchFlex contract, the `__reserved` storage array for future upgrades is declared in the middle of the contract's storage variables, specifically between `managedReporters` and `timeOfLastDistribution`.

According to best practices for upgradeable contracts, reserved storage slots should be placed at the end of all storage variable declarations to maintain a clean and organized storage layout.

While this does not affect current functionality, placing reserved slots in between storage variables makes the contract's storage layout less maintainable and could lead to confusion and potential issues when implementing future upgrades.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:L/D:N/Y:N (1.3)

Recommendation

Move the `__reserved` array declaration to the end of all storage variable declarations, after `timeOfLastDistribution`. This maintains a clearer separation between active storage variables and reserved slots for future use.

Remediation Comment

ACKNOWLEDGED: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"We understand that current placement is not in line with best practice placement (at the end of declarations). Given there is no impact on functionality we will leave the variable placement as is for now."

References

fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L43

9.2 MANAGEDQUERY BECOMES PERMANENT AFTER FIRST VALUE SUBMISSION

// INFORMATIONAL

Description

The `removeManagedQuery()` function in FetchFlex has a limitation: it can only be executed before any values are submitted to the query feed. This is because the function requires that `reports[_queryId].timestamps.length == 0`, but the `timestamps` array only grows and is never cleared throughout the contract's lifecycle.

This creates a permanent state where managed queries cannot be removed once they become active, as the first value submission permanently blocks the removal functionality. While this could be an intentional security feature to prevent tampering with historical data, it's not clearly documented and could lead to governance confusion.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (0.6)

Recommendation

If the current behavior is intentional, update the contract to clearly document this limitation in the NatSpec comments of the `removeManagedQuery()` function, explaining that managed queries become permanent once the first value is submitted.

To allow removal of managed queries after initial use, consider implementing a deprecation system where queries can be first marked as inactive (preventing new submissions) and then removed after a safety timelock period to provide flexibility for managing outdated feeds while preserving historical data integrity.

Remediation Comment

ACKNOWLEDGED: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"This is the intended design since once a query is set as a managed feed and a price is reported to it, the query id must remain as a managed feed to avoid any misuse or misunderstanding with the community (non-managed feed) reporters."

References

fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L945-L953

9.3 MISSING ERROR DESCRIPTIONS

// INFORMATIONAL

Description

The `onlyManager()` modifier in the `FetchFlex` contract reverts if the `msg.sender` is not the manager if the `queryId` provided, but the error message is not provided.

Similarly, in the `_updateStakeAndPayRewards()` function of the same contract, the function fails when the token transfer fails but an error message is not provided.

This can make it difficult for users and developers to understand why a transaction failed and how to resolve the issue.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add an error message to provide more information about the reason for the revert.

Remediation Comment

FUTURE RELEASE: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"We have built off chain tooling that integrates with the contract and provides sufficient end user error messages. However, we may add revert messages directly in the contract in a future upgrade."

References

fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L74-L77
fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L1088

9.4 INCORRECT NATSPEC DOCUMENTATION

// INFORMATIONAL

Description

The `verify()` function in the `FetchFlex` contract has a misleading NatSpec documentation. The comment states "return bool value used to verify valid Fetch contracts", but the function actually returns a `uint256` value.

This inconsistency between documentation and implementation could mislead developers integrating with or maintaining the contract.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Update the NatSpec documentation to accurately reflect the function's return type and purpose.

Remediation Comment

FUTURE RELEASE: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"We will update the comments to accurately reflect the function's return type and purpose in a future upgrade."

References

fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L924

9.5 MISSING EVENTS

// INFORMATIONAL

Description

In the `FetchFlex` contract, there are instances where administrative functions change contract state by modifying core state variables. However, these changes are not reflected in any event emission.

Instances of this issue can be found in:

- `FetchFlex.setupManagedQuery()`
- `FetchFlex.removeManagedQuery()`
- `FetchFlex.addReporter()`
- `FetchFlex.removeReporter()`

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Emit events for all state changes that occur as a result of administrative functions to facilitate off-chain monitoring of the system.

Remediation Comment

FUTURE RELEASE: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"We will emit events for all state changes that occur as a result of administrative functions to facilitate off-chain monitoring of the system in a future upgrade."

References

fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L936-L967

9.6 CONSIDER USING FLEXIBLE PRAGMA FOR CONTRACT INTERFACES

// INFORMATIONAL

Description

Several interface contracts in the codebase use strictly locked pragma versions (`0.8.3`) or overly flexible (`>= 0.8.0`).

While locked pragma versions are a best practice for implementation contracts, for interfaces that are meant to be widely reused, a more flexible pragma version (within the major version, e.g. `0.8`) allows consuming contracts to implement the interface regardless of their fixed Solidity version.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider using caret pragma (`^`) for interfaces to specify compatibility while allowing minor version updates:

```
pragma solidity ^0.8.0;
```

For more reference, see this discussion or this recommendation.

Remediation Comment

FUTURE RELEASE: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"We will update interface contract pragma in a future upgrade."

References

fetchoracle/fetch-contracts/contracts/interfaces/IERC20.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IERC2362.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IFetch.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IFetchFlex.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IFetchToken.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IMappingContract.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IOracle.sol#L2
fetchoracle/fetch-contracts/contracts/interfaces/IQueryDataStorage.sol#L2

9.7 UNUSED FUNCTION

// INFORMATIONAL

Description

The `getQueryConfig()` function of the `FetchFlex` contract is declared with the `internal` visibility modifier but it is not called within the contract.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider updating the visibility of the `getQueryConfig()` function to `external` if it is intended to be called by other contracts or users. Alternatively, call the function from the `submitValue()` function replacing the redundant code to verify if the `queryId` requires staking.

Remediation Comment

FUTURE RELEASE: The Fetch Oracle team made a business decision to acknowledge this finding and not alter the contracts, stating that:

We will update function visibility to external in a future upgrade.

References

fetchoracle/fetch-contracts/contracts/FetchFlex.sol#L993-L997

9.8 MODIFIED OPENZEPPELIN CONTRACTS

// INFORMATIONAL

Description

All files in the `dependencies/utils` directory contain modifications from the original OpenZeppelin contracts (upgradeable contracts are using version `4.9.3` according to `package.json` but `EnumerableSet` is taken from version > 5.0.0).

While these changes appear not to introduce any vulnerabilities, it is important to note that the contracts have been modified, introducing inconsistencies with widely tested standard implementations.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider using the original OpenZeppelin contracts without modifications to reduce the risk of vulnerabilities or bugs being introduced.

Additionally, consider replacing the `TransferHelper` contract with the `SafeERC20` contract from OpenZeppelin to ensure the safe transfer of ERC20 tokens with dependencies consistency across the codebase.

Remediation Comment

FUTURE RELEASE: The Fetch Oracle **team** made a business decision to acknowledge this finding and not alter the contracts, stating that:

"We will update OpenZeppelin contracts to v5.2.x and use installed dependencies using NPM in a future upgrade."

References

fetchoracle/fetch-contracts/tree/3fbd55fe6932a9b386750fc357f156854bd0ed68
