Guild Audits

JULY 29TH, 2025

IFAORACLE FINAL AUDIT REPORT

By Guild Audits

TABLE OF CONTENTS

DISCLAIMER

The Guild Audit team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

EXECUTIVE SUMMARY

This section will represent the summary of the whole audit.

PROJECT AUDIT SCOPE AND FINDINGS

The motive of this audit is to review the codebase of Coinbase contracts for the purpose of achieving secured, correctness and quality smart contracts.
Number of Contracts in Scope: All the contract (1547 SLOC)
Duration for Audit: 14 days

Vulnerability Summary:
Total issues: 7

Low: 5
Informational: 2

FINDINGS

Index	Title	Severity	Status of Issues
01	[L-01] Inefficient Stale Check Allows Redundant Price Updates	Low	Resolved
02	[L-02] Self-Pairing Allowed in Price Calculation	Low	Resolved
03	[L-03] Lack of Zero-Price Validation in Price Submissions	Low	Resolved
04	[L-04] Silent Reverts in Internal Functions Obscure Failure Root Cause	Low	Resolved
05	[L-05] Indexing a Struct Provides No Searchable Value	Low	Resolved
06	[I-01] Incorrect Error Reporting in Batch Pair Functions	Informational	Resolved
07	[I-02] Unnecessary Comments and Inefficient Logic in _getAssetInfo	Informational	Resolved

MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follows:
Manual Review: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
Automated Testing: This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools used are Slither, Echidna and others.
Functional Testing: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to exploit the contracts.

The methodologies for establishing severity issues:

- High Level Severity Issues
- Medium Level Severity Issues
- Low Level Severity Issues
- Informational Level Severity Issues

TYPES OF SEVERITY

Every issue in this report has been assigned a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

These are critical issues that can lead to a significant loss of funds, compromise of the contract's integrity, or core function of the contract not working. Exploitation of such vulnerabilities could result in immediate and catastrophic consequences, such as:

- Complete depletion of funds.
- Permanent denial of service (DoS) to contract functionality.
- Unauthorized access or control over the contract

Medium Severity Issues

These issues pose a significant risk but require certain preconditions or complex setups to exploit. They may not result in immediate financial loss but can degrade contract functionality or pave the way for further exploitation. Exploitation of such vulnerabilities could result in partial denial of service for certain users,leakage of sensitive data or unintended contract behavior under specific circumstances.

Low Severity Issues

These are minor issues that have a negligible impact on the contract or its users. They may affect efficiency, readability, or clarity but do not compromise security or lead to financial loss. Impacts are minor degradation in performance,confusion for developers or users interacting with the contract and low risk of exploitation with limited consequences.

Informational

These are not vulnerabilities in the strict sense but observations or suggestions for improving the contract's code quality, readability, and adherence to best practices.

There is no direct impact on the contract’s functionality or security, it is aimed at improving code standards and developer understanding.

TYPES OF ISSUES

Open
Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved
These are the issues identified in the audit and have been successfully fixed.

Acknowledged
Vulnerabilities that have been acknowledged but are yet to be resolved.

Partially Resolved
Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

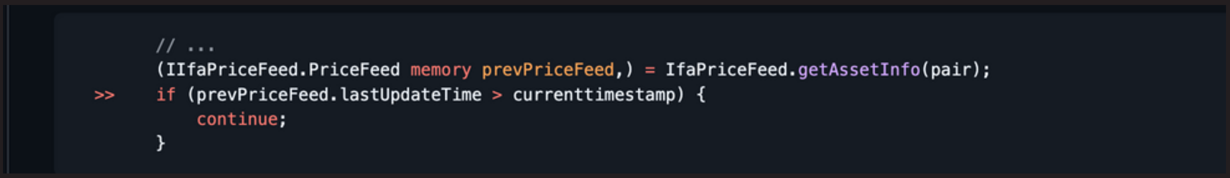
LOW SEVERITY ISSUE

[L-01] Inefficient Stale Check Allows Redundant Price Updates

Summary:

The submitPriceFeed function in *IfaPriceFeedVerifier.sol* uses a strict greater-than (>) comparison when checking for stale price data. This logic incorrectly allows a relayer to submit a price update with a timestamp that is equal to the one already stored on-chain. This triggers a redundant state change for no functional benefit. The check should use a *greater-than-or-equal-to (>=)* comparison to prevent these unnecessary updates. The core of the issue lies in the if condition within the for loop of the submitPriceFeed function.

File: oracle_contract-main/src/IfaPriceFeedVerifier.sol



```
        // ...
        (IIfaPriceFeed.PriceFeed memory prevPriceFeed,) = IfaPriceFeed.getAssetInfo(pair);
>>      if (prevPriceFeed.lastUpdateTime > currenttimestamp) {
            continue;
        }
```

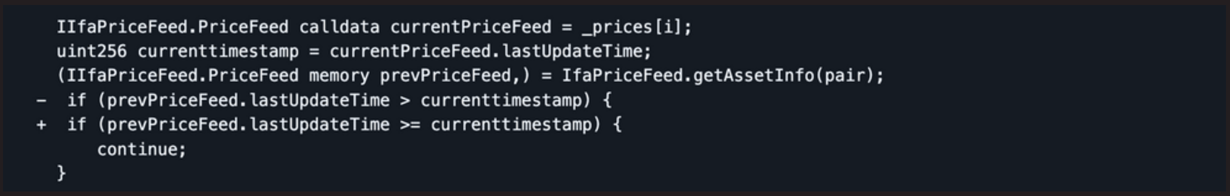
When prevPriceFeed.lastUpdateTime is equal to currenttimestamp, the condition *prevPriceFeed.lastUpdateTime* > *currenttimestamp* evaluates to false.

Impact

If a relayer re-submits price data with the same timestamp as the existing on-chain data (which can happen during network retries or normal operational redundancy), the contract will perform an unnecessary state update.

Recommendation

We recommend changing the strict greater-than (>) comparison to a greater-than-or-equal-to (>=) comparison. This will ensure that the contract only processes and stores price feeds that have a strictly newer timestamp, preventing unnecessary updates.



```
    IIfaPriceFeed.PriceFeed calldata currentPriceFeed = _prices[i];
    uint256 currenttimestamp = currentPriceFeed.lastUpdateTime;
    (IIfaPriceFeed.PriceFeed memory prevPriceFeed,) = IfaPriceFeed.getAssetInfo(pair);
-   if (prevPriceFeed.lastUpdateTime > currenttimestamp) {
+   if (prevPriceFeed.lastUpdateTime >= currenttimestamp) {
        continue;
    }
```

Status: **Resolved**

LOW SEVERITY ISSUE

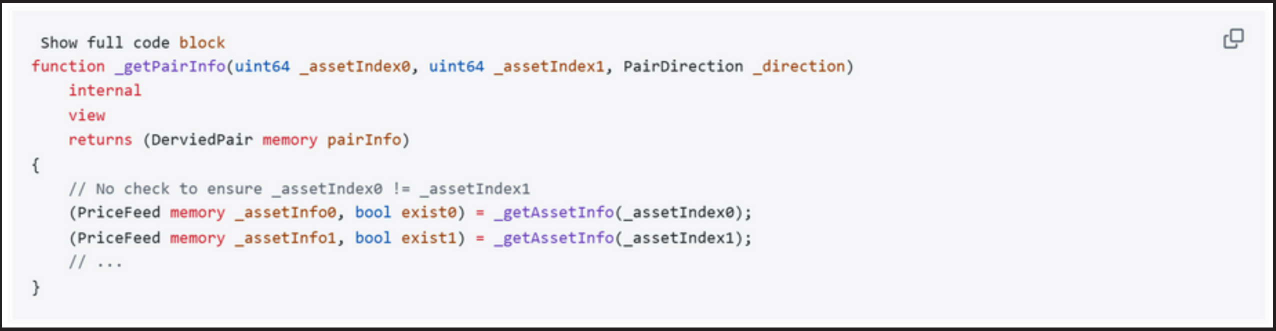
[L-02] Self-Pairing Allowed in Price Calculation

Summary

The internal function *_getPairInfo*, which is used by all public-facing pair-fetching functions (*getPairbyId*, *getPairsbyIdForward*, etc.), does not validate that the two asset indexes provided (*_assetIndex0* and *_assetIndex1*) are different. This allows a caller to request the price for a pair of the same asset (e.g., BTC/BTC). The function will not revert but will return a derived price of exactly 1.0, which could lead to unexpected behavior or manipulation in protocols that consume this oracle data.

The *_getPairInfo* function in *IfaPriceFeed.sol* is the core logic for calculating derived exchange rates. It accepts two asset indexes but lacks a check to ensure they are not identical.

File: oracle_contract-main/src/IfaPriceFeed.sol



```
Show full code block
function _getPairInfo(uint64 _assetIndex0, uint64 _assetIndex1, PairDirection _direction)
    internal
    view
    returns (DerviedPair memory pairInfo)
{
    // No check to ensure _assetIndex0 != _assetIndex1
    (PriceFeed memory _assetInfo0, bool exist0) = _getAssetInfo(_assetIndex0);
    (PriceFeed memory _assetInfo1, bool exist1) = _getAssetInfo(_assetIndex1);
    // ...
}
```

When _assetIndex0 is the same as _assetIndex1, the function will fetch the same price data for both assets. The subsequent calculation will result in:

decimal: MAX_DECIMAL_NEGATIVE

lastUpdateTime: The timestamp of that single asset.

*derivedPrice: 1 * 10**18 (representing a price of 1.0)*

Impact

While not a direct exploit, it opens a potential attack vector. If a consuming protocol has a vulnerability that is triggered by a price of exactly 1.0, an attacker could use this behavior to their advantage. For example, a flawed rewards or fee mechanism might behave differently for trades at a 1:1 ratio. It is also logically inconsistent for a price feed oracle to provide a derived price for an asset against itself. This indicates a lack of robustness in handling edge cases.

Recommendation

It is recommended to add a validation check at the beginning of the *_getPairInfo* function to ensure the two asset indexes are not the same. This makes the function's behavior more robust, predictable, and secure.

Status: **Resolved**

LOW SEVERITY ISSUE

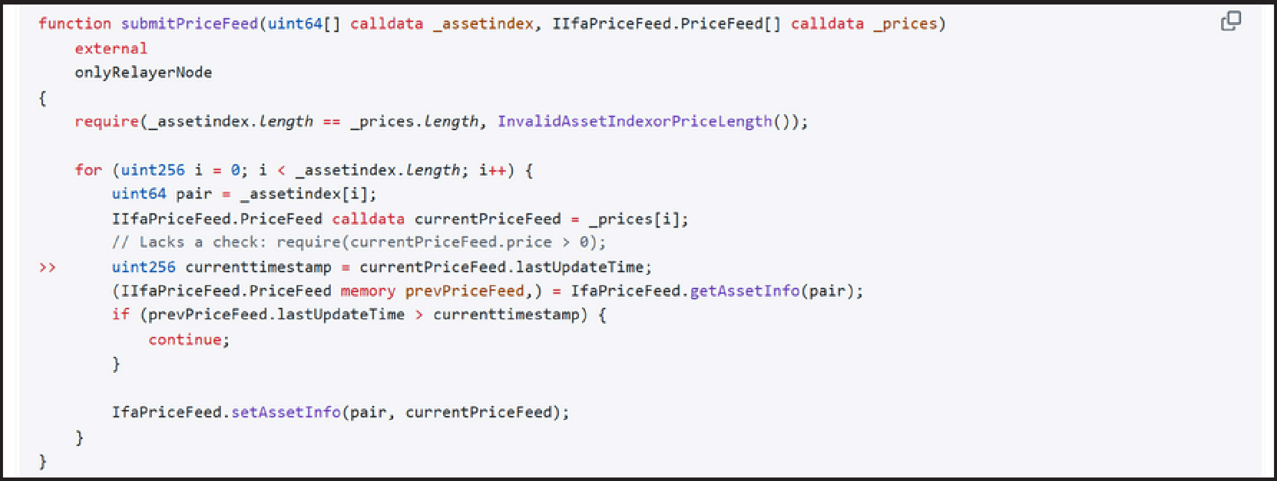
[L-03] Lack of Zero-Price Validation in Price Submissions

Summary

The submitPriceFeed function in ***IfaPriceFeedVerifier.sol*** does not validate that submitted prices are greater than zero. The relayer could provide a price of 0 for an asset, which would be accepted and stored on-chain. This creates a Denial of Service (DoS) vulnerability for any function that attempts to calculate a derived pair where the zero-priced asset is the denominator, as it will lead to a division-by-zero error.

The *submitPriceFeed* function is the entry point for all new price data. While it checks for stale data using lastUpdateTime, it completely omits a check to ensure the price field itself is a valid, non-zero number.

File: oracle_contract-main/src/IfaPriceFeedVerifier.sol



```
function submitPriceFeed(uint64[] calldata _assetindex, IIfaPriceFeed.PriceFeed[] calldata _prices)
    external
    onlyRelayerNode
{
    require(_assetindex.length == _prices.length, InvalidAssetIndexorPriceLength());

    for (uint256 i = 0; i < _assetindex.length; i++) {
        uint64 pair = _assetindex[i];
        IIfaPriceFeed.PriceFeed calldata currentPriceFeed = _prices[i];
        // Lacks a check: require(currentPriceFeed.price > 0);
>>      uint256 currenttimestamp = currentPriceFeed.lastUpdateTime;
        (IIfaPriceFeed.PriceFeed memory prevPriceFeed,) = IfaPriceFeed.getAssetInfo(pair);
        if (prevPriceFeed.lastUpdateTime > currenttimestamp) {
            continue;
        }

        IfaPriceFeed.setAssetInfo(pair, currentPriceFeed);
    }
}
```

Impact

If an asset's price is updated to 0, any call to getPairbyId or related functions where this asset is the denominator will fail. The *_getPairInfo* function uses *FixedPointMathLib.mulDiv,* which correctly reverts on division by zero. This will cause transactions to fail, potentially halting critical operations like liquidations, swaps, or borrowing in dependent DeFi protocols.

Recommendation
It is essential to add a strict validation check in submitPriceFeed to ensure that all submitted prices are greater than zero. This can be implemented by adding a require statement and a corresponding custom error for clarity.

Status: **Resolved**

LOW SEVERITY ISSUE

[L-04] Silent Reverts in Internal Functions Obscure Failure Root Cause

Summary

In the ***IfaPriceFeed.sol*** contract, within the ***_scalePrice*** function, two require statements are used to validate the scaled price but the require statements have no error message.

```
function _scalePrice(int256 price, int8 decimal) public pure returns (uint256) {
  uint256 scalePrice = uint256(price) * 10 ** (MAX_DECIMAL - abs(decimal));
  require(scalePrice <= MAX_INT256); <<-@audit reverts without an error msg
  require(scalePrice > uint256(price)); <<-@audit reverts without an error msg
  return scalePrice;
}
```

Impact

Since ***_scalePrice*** is an internal function and is only called within another internal function, ***_getPairInfo***, its two require statements do not include error messages which means if there's a revert inside the ***_scalePrice*** function, the caller will receive no context making it difficult to identify the root cause of the failure. While this does not present a direct security vulnerability, it significantly affects code clarity and debuggability for developers directly interacting with functions like ***getPairsbyIdBackward, getPairsbyId, getPairsbyIdForward.***

Recommendation

Add error messages to both require methods.

```
 function _scalePrice(int256 price, int8 decimal) public
      pure returns (uint256) {
       uint256 scalePrice = uint256(price) * 10 ** (MAX_DECIMAL - abs(decimal));
       require(scalePrice <= MAX_INT256, "Scaled price exceeds max int256");
       require(scalePrice > uint256(price), "Scaled price is not greater than
original price");
 return scalePrice;
}
```

Status: **Resolved**

LOW SEVERITY ISSUE

[L-06] Indexing a Struct Provides No Searchable Value

Summary

The ***AssetInfoSet*** event, defined in the IIfaPriceFeed.sol interface, incorrectly marks the PriceFeed struct as indexed. In Solidity, indexing a struct does not make its individual fields searchable. Instead, the entire struct is ABI-encoded and hashed, and this single bytes2 hash is stored as the indexed topic. This is not useful for off-chain filtering, misleads developers about the event's capabilities.

```
event AssetInfoSet(uint64 indexed _assetIndex, PriceFeed
indexed assetInfo);
```

When an event parameter of a complex type (like a struct or array) is marked as indexed, Solidity computes the keccak256 hash of its ABI-encoded representation and stores that hash as the topic.
This means that an off-chain service trying to filter for these events would see a topic like 0x123...abc for the assetInfo. To find a specific event, the service would need to know the exact contents of the PriceFeed struct (decimal, lastUpdateTime and price) to reconstruct the hash and search for it. This defeats the purpose of filtering, which is to find events based on variable criteria (e.g., "find all price updates where the price was above $50,000").
Check sample report here

Impact

The indexed keyword on the struct parameter suggests a search capability that does not exist. This can lead to incorrect assumptions and wasted effort by developers who try to build services that rely on filtering these events by their content.

Recommendation

To resolve this, the indexed keyword should be removed from the assetInfo parameter in the event definition. The full struct data will still be available in the non-indexed (data) portion of the event log, where it can be easily decoded and accessed by off-chain services. This change will make the event's capabilities clear and unambiguous.

Status: Resolved

INFORMATIONAL SEVERITY ISSUE

[I-01] Incorrect Error Reporting in Batch Pair Functions

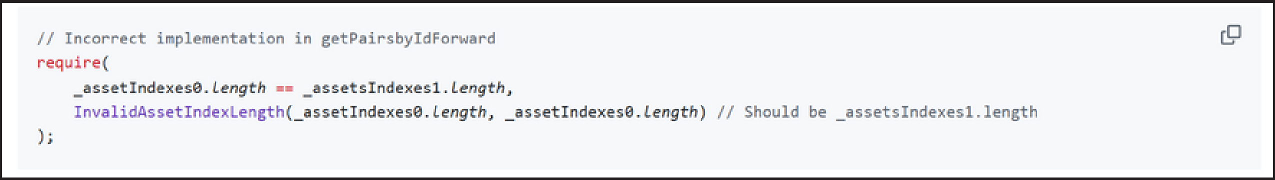
Summary

The ***getPairsbyIdForward***, ***getPairsbyIdBackward***, and ***getPairsbyId*** functions in IfaPriceFeed.sol provide incorrect details when they revert due to mismatched input array lengths. The custom errors are populated with the length of the first array for both length arguments, which hides the actual mismatched length of the second array from the caller and complicates debugging.

The contract includes several functions for fetching multiple pair prices in a single call. These functions correctly validate that the input arrays (*_assetIndexes0, _assetsIndexes1*, and sometimes _direction) have equal lengths. However, when this check fails, the arguments passed to the custom error are incorrect.

getPairsbyIdForward and ***getPairsbyIdBackward***

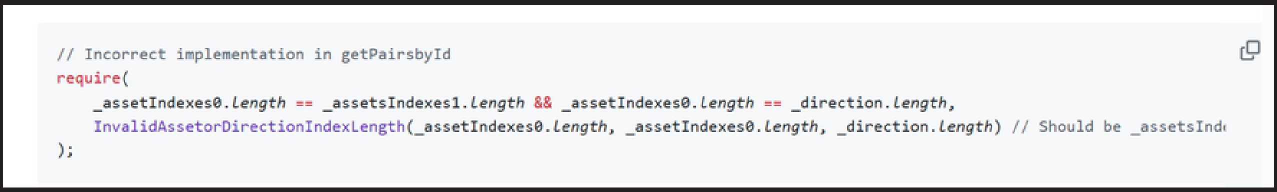
These functions use the InvalidAssetIndexLength error, which is defined to accept two uint256 parameters: the length of the first array and the length of the second. The implementation incorrectly passes _assetIndexes0.length for both.



```
// Incorrect implementation in getPairsbyIdForward
require(
    _assetIndexes0.length == _assetsIndexes1.length,
    InvalidAssetIndexLength(_assetIndexes0.length, _assetIndexes0.length) // Should be _assetsIndexes1.length
);
```

getPairsbyId

This function has a similar issue with the InvalidAssetorDirectionIndexLength error. It passes *_assetIndexes0.length* as the value for the second asset array's length instead of the correct *_assetsIndexes1.length*.



```
// Incorrect implementation in getPairsbyId
require(
    _assetIndexes0.length == _assetsIndexes1.length && _assetIndexes0.length == _direction.length,
    InvalidAssetorDirectionIndexLength(_assetIndexes0.length, _assetIndexes0.length, _direction.length) // Should be _assetsInd
);
```

Impact

When a developer calls one of these functions with mismatched array lengths, the revert message will be misleading. For example, if _assetIndexes0 has a length of 5 and _assetsIndexes1 has a length of 4, the error will report the lengths as (5, 5) instead of the correct (5, 4). This makes it significantly harder for developers to debug their integration, as the error message does not accurately reflect the state that caused the transaction to fail.

Recommendation

The arguments passed to the custom errors should be corrected to reflect the actual lengths of the input arrays. This provides accurate and helpful debugging information to the caller.

Status: **Resolved**

INFORMATIONAL SEVERITY ISSUE

[I-02] Unnecessary Comments and Inefficient Logic in _getAssetInfo

Summary

The *_getAssetInfo* function in IfaPriceFeed.sol contains commented-out code and non-standard comments that reduce code clarity.

```
//require(_assetInfo[_assetIndex].lastUpdateTime > 0,
InvalidAssetIndex(_assetIndex));
//...
```

Impact

This is an informational finding. While there is no direct security vulnerability, the issues affect code quality. The unnecessary comments make the code harder to read and maintain.

Recommendation

It is recommended to refactor the *_getAssetInfo* function to remove the comments.

Status: **Resolved**