// Private Smart Contract Security Assessment 03.25.2024 - 04.14.2024

NLX
CoreDAO

HALBORN

NLX - CoreDAO

Prepared by: HALBORN

Last Updated 04/20/2024

Date of Engagement by: March 25th, 2024 - April 14th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	1	1	0	1

TABLE OF CONTENTS

1. Introduction

The CoreDAO team engaged Halborn to conduct a security assessment on their smart contracts beginning on *03/25/2024* and ending on *04/15/2024*. The security assessment was scoped to the smart contracts provided in the GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 2.5 weeks 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 experts 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 security that were successfully addressed by the CoreDAO team.

3. Test Approach And Methodology

Halborn performed a combination of manual 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 the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contracts, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

Out-Of-Scope

- External libraries and financial-related attacks.
- External GMX code vulnerabilities.
- New features/implementations after/with the **remediation commit IDs.**
- Changes that occur outside the scope of PRs.

4. 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.

4.1 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:

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
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:

$$E = \prod m_e$$

4.2 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:

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

4.3 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

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: nlx-synthetics

(b) Assessed Commit ID: a9fbd4b

(c) Contracts in scope:

- oracle/Oracle.sol
- utils/MulticallWithValue.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- c385344
- a2f22a0

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	1	0	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-03 - NON-REFUNDED EXCESS FEE IN _SETPRICESFROMPRICEFEED FUNCTION	High	SOLVED - 04/18/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - MULTIPLE PRICE FETCHES LEADING TO ARBITRAGE IN ORACLE IMPLEMENTATION	Medium	SOLVED - 04/18/2024
HAL-02 - INCONSISTENCY IN FEE CALCULATION UPDATE ACROSS CONTRACT IMPLEMENTATIONS	Informational	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 (HAL-03) NON-REFUNDED EXCESS FEE IN _SETPRICESFROMPRICEFEED FUNCTION

// HIGH

Description

The `_setPricesFromPriceFeeds` function in the smart contract does not refund the excess fee to the sender if the amount sent (`msg.value`) exceeds the actual update fee (`updateFee`) required by the `pyth` price feed update.
The function `_setPricesFromPriceFeeds` is designed to update price feeds using an external oracle (Pyth network). The function retrieves the update fee using `pyth.getUpdateFee(pythUpdateData)` and checks if the `msg.value` provided by the sender covers this fee with the requirement `require(updateFee <= msg.value, "not enough funds to update price feeds");`. However, if `msg.value` exceeds `updateFee`, the surplus is not refunded back to the sender, effectively causing the sender to lose the excess amount.

```
    // @dev set prices using external price feeds to save costs for tokens with stable
prices
    // @param dataStore DataStore
    // @param eventEmitter EventEmitter
    // @param tokens the tokens to set the prices using the price feeds for
    function _setPricesFromPriceFeeds(DataStore dataStore, EventEmitter eventEmitter,
address[] memory tokens, bytes[] memory pythUpdateData) internal {
        uint updateFee = pyth.getUpdateFee(pythUpdateData);
        require(updateFee <= msg.value, "not enough funds to update price feeds");

        pyth.updatePriceFeeds{value: updateFee}(pythUpdateData);
...
}
```

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "ds-test/test.sol";
import "../src/PriceFeedUpdater.sol";
import "../src/Pyth.sol";
import "../src/EventEmitter.sol";
```

```
contract PriceFeedUpdaterTest is DSTest {
    PriceFeedUpdater updater;
    Pyth pyth;
    EventEmitter eventEmitter;
    address[] tokens;

    function setUp() public {
        pyth = new Pyth();
        eventEmitter = new EventEmitter();
        updater = new PriceFeedUpdater(address(pyth), address(eventEmitter));
    }

    function testExcessFeeRefund() public {
        uint sentAmount = 0.02 ether;
        uint updateFee = 0.01 ether;

        uint balanceBefore = address(this).balance;
        updater._setPricesFromPriceFeeds{value: sentAmount}(tokens, new bytes(0));
        uint balanceAfter = address(this).balance;

        assertEq(balanceBefore - balanceAfter, updateFee, "Excess fee was not
refunded");
    }
}
St
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:C/R:P/S:C (7.0)

Recommendation

Implement a refund mechanism within the `_setPricesFromPriceFeeds` function to return any excess amount over the `updateFee` back to the sender.

Remediation Plan

SOLVED : The **CoreDAO team** solved the issue by adding refund logic on the contracts.

Remediation Hash

https://github.com/NLX-Protocol/nlx-synthetics/commit/c3853446a41e2e963e087f1c55ac935bfaeaa1a3

7.2 (HAL-01) MULTIPLE PRICE FETCHES LEADING TO ARBITRAGE IN ORACLE IMPLEMENTATION

// MEDIUM

Description

In the current implementation of the `_setPricesFromPriceFeeds` function within the contract, there exists a vulnerability associated with fetching multiple prices from the Pyth network oracle in the same transaction. This vulnerability could potentially be exploited to conduct risk-free arbitrage, adversely affecting other users.

The contract uses the Pyth network oracle to fetch and update price feeds. The Pyth network updates price data every 400ms, making it highly dynamic. The `updatePriceFeeds` function is called with price update data (`pythUpdateData`), and for each call, the oracle may return a different price based on the most recent data submitted to the network. Since a single transaction can involve multiple calls to this function with different `pythUpdateData`, it's possible to fetch different prices within the same transaction.

This behavior can be exploited to create arbitrage opportunities by manipulating trade orders based on the fetched prices.

Proof of Concept

Evidence shows that it is feasible to submit and retrieve two distinct prices within the same transaction. Within this transaction, the `updatePriceFeeds` function is invoked using two separate prices. Following each invocation, the prevailing price is retrieved, and an event is broadcast that includes both the price and the timestamp. Clearly, the prices obtained from each price query differ.

```
Address 0x8250f4af4b972684f7b336503e2d6dfedeb1487a
Name    PriceFeedUpdate (index_topic_1 bytes32 id, uint64 publishTime,
int64 price, uint64 conf)
Topics  0
0xd06a6b7f4918494b3719217d1802786c1f5112a6c1d88fe2cfec00b4584f6aec
        1 FF61491A931112DDF1BD8147CD1B641375F79F5825126D665480874634FD0ACE
Data    publishTime: 1706358779
        price: 226646416525
        conf: 115941591

Address 0xbf668dadb9cb8934468fcba6103fb42bb50f31ec
Topics  0
0x734558db0ee3a7f77fb28b877f9d617525904e6dad1559f727ec93aa06370866
Data    226646416525
        1706358779

Address 0x8250f4af4b972684f7b336503e2d6dfedeb1487a
Name    PriceFeedUpdate (index_topic_1 bytes32 id, uint64 publishTime,
```

```
int64 price, uint64 conf)View Source
Topics  0
0xd06a6b7f4918494b3719217d1802786c1f5112a6c1d88fe2cfec00b4584f6aec
        1 FF61491A931112DDF1BD8147CD1B641375F79F5825126D665480874634FD0ACE
Data    publishTime: 1706358790
        price: 226649088828
        conf: 119840116

Address 0xbf668dadb9cb8934468fcba6103fb42bb50f31ec
Topics  0
0x734558db0ee3a7f77fb28b877f9d617525904e6dad1559f727ec93aa06370866
Data    226649088828
        1706358790
```

BVSS

AO:A/AC:L/AX:L/C:M/I:H/A:N/D:N/Y:N/R:P/S:C (5.5)

Recommendation

Adding checks to ensure that the price used in trades does not deviate significantly within short time frames, particularly within the same block.

Remediation Plan

SOLVED : The **CoreDAO team** solved the issue by implementing suggested recommendations.

Remediation Hash

https://github.com/NLX-Protocol/nlx-synthetics/commit/a2f22a08fa824131d9f88fdff1e23684ad84b0db

7.3 (HAL-02) INCONSISTENCY IN FEE CALCULATION UPDATE ACROSS CONTRACT IMPLEMENTATIONS

// INFORMATIONAL

Description

An update to the fee calculation logic was made in the `ReaderUtils.sol` file of the GMX code base, but this change has not been reflected in the NLX code implementation. The updated logic includes a revised calculation for total cost amounts excluding and including funding fees.
The GMX implementation has added a new calculation method for total cost amounts, which segregates the total cost calculation into excluding and including funding fees. Specifically, the change involves adding the funding fee to the previously calculated total cost amount excluding funding, thereby providing a more granular breakdown of fees.

Score

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

Recommendation

Review and update the NLX code to align with the changes made in the GMX implementation. Ensure that all related parts of the system use a consistent method for fee calculation.

Remediation Plan

ACKNOWLEDGED: The **CoreDAO team** acknowledged this finding.

8. AUTOMATED TESTING

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.
All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.
