// Private Smart Contract Security Assessment 04.15.2024 - 05.03.2024

Ecosystem - Colend Protocol

CoreDAO

HALBORN

Ecosystem - Colend Protocol - CoreDAO

Prepared by: HALBORN

Last Updated 09/17/2024

Date of Engagement by: April 15th, 2024 - May 3rd, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	0	0	0	1	3

TABLE OF CONTENTS

1. Introduction

`The CoreDAO team` engaged Halborn to conduct a security assessment on their smart contracts beginning on *04/15/2024* and ending on *04/30/2024*. 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 3 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 acknowledged by the `Colend 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`).

3.1 Out-Of-Scope

- External libraries and financial-related attacks.
- External AAVE V3 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
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: aave-v3-core

(b) Assessed Commit ID: dad86c8

(c) Items in scope:

- contracts/protocol/tokenization/base/ScaledBalanceTokenBase.sol
- contracts/protocol/libraries/math/WadRayMath.sol
- contracts/misc/AaveOracle.sol
- contracts/protocol/tokenization/StableDebtToken.sol

Out-of-Scope: Forked code

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	1	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT CHECK FOR VALUE IN MAPPING LEADS TO INCORRECT CALL TO PYTH ORACLE	LOW	RISK ACCEPTED
NEW EMPTY MARKETS ARE VULNERABLE TO PRICE MANIPULATION	INFORMATIONAL	ACKNOWLEDGED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
ENHANCE AAVEORACLE IMPLEMENTATION FOR FUTURE COMPATIBILITY WITH OTHER ORACLES	INFORMATIONAL	ACKNOWLEDGED
DISABLE BORROWING ON THE STABLEDEBT TOKENS	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 INCORRECT CHECK FOR VALUE IN MAPPING LEADS TO INCORRECT CALL TO PYTH ORACLE

// LOW

Description

While reviewing the `Pyth Oracle's` implementation within the `AaveOracle.sol` contract, an issue was discovered with **the second if statement** in the `getAssetPrice(address asset)` function.
The statement incorrectly compares the `priceFeedId`, a variable of type bytes32, with an integer value of `0`.
Consequently, when the required asset does not correspond to a key in the `priceFeedIds` mapping, the contract will perform invalid, reverting calls to the `Pyth Oracle` using the **default value for bytes32,** because the function execution will never be halted in case of inexistent or invalid `priceFeedId`, because `priceFeedId` can never be equal to `0`.
- contracts/misc/AaveOracle.sol [Lines: 102-114]

```
102 | function getAssetPrice(address asset) public view override returns (uint2
103 |     bytes32 priceFeedId = priceFeedIds[asset];
104 |     if (asset == BASE_CURRENCY) return BASE_CURRENCY_UNIT;
105 |     if (priceFeedId == 0) return 0;
106 |     PythStructs.Price memory priceStruct = IPyth(ADDRESSES_PROVIDER.getPy
107 |       priceFeedId
108 |     );
109 |     return uint256(uint64(priceStruct.price));
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to modify the second if statement, so the comparison is performed between two elements of the same type - bytes32, and therefore the function will **return 0**, in case of invalid or non-existent `priceFeedId` for the required asset, effectively halting the function execution before the external call to the `Pyth` contract.

```
    if (priceFeedId == bytes32(0)) return 0;
```

Remediation

RISK ACCEPTED: The **Colend team** accepted the risk of the issue.

7.2 NEW EMPTY MARKETS ARE VULNERABLE TO PRICE MANIPULATION

// INFORMATIONAL

Description

The root cause of this known vulnerability is the loss of precision during smart contract operations, which can lead to price manipulation of the underlying assets.
Specifically, when a new market is activated in the lending protocol, there is a time window that can be exploited by an attacker. During this time window, the attacker can be the first individual to supply funds in the new market, manipulating the `liquidityIndex`, a key factor in determining user balances. The manipulation of the `liquidityIndex` allows the attacker to borrow all the underlying assets, resulting in a significant loss of funds.

- contracts/protocol/tokenization/base/ScaledBalanceTokenBase.sol [Lines: 99-120]

```
99    function _burnScaled(address user, address target, uint256 amount, uint
100     uint256 amountScaled = amount.rayDiv(index);
101     require(amountScaled != 0, Errors.INVALID_BURN_AMOUNT);
102
103     uint256 scaledBalance = super.balanceOf(user);
104     uint256 balanceIncrease = scaledBalance.rayMul(index) -
105       scaledBalance.rayMul(_userState[user].additionalData);
106
107     _userState[user].additionalData = index.toUint128();
108
109     _burn(user, amountScaled.toUint128());
110
111     if (balanceIncrease > amount) {
112       uint256 amountToMint = balanceIncrease - amount;
113       emit Transfer(address(0), user, amountToMint);
114       emit Mint(user, user, amountToMint, balanceIncrease, index);
115     } else {
116       uint256 amountToBurn = amount - balanceIncrease;
117       emit Transfer(user, address(0), amountToBurn);
118       emit Burn(user, target, amountToBurn, balanceIncrease, index);
119     }
120   }
```

The vulnerability is further exacerbated by a known rounding issue in the `rayDiv` function, which is used to divide two ray values. The flaw in the `rayDiv` function can be taken advantage of to siphon funds from the pool, resulting in a cumulative precision loss that is magnified with repeated deposit and withdrawal operations.

- contracts/protocol/libraries/math/WadRayMath.sol [Lines: 83-92]

```
83    function rayDiv(uint256 a, uint256 b) internal pure returns (uint256 c)
84      // to avoid overflow, a <= (type(uint256).max - halfB) / RAY
85      assembly {
86        if or(iszero(b), iszero(iszero(gt(a, div(sub(not(0), div(b, 2)), RA
87          revert(0, 0)
88        }
89
90        c := div(add(mul(a, RAY), div(b, 2)), b)
91      }
92    }
```

Score

Impact:

Likelihood:

Recommendation

Consider minting minimum **1e6** token to the dead address.

Remediation

ACKNOWLEDGED: The **Colend team** acknowledged the issue.

7.3 ENHANCE AAVEORACLE IMPLEMENTATION FOR FUTURE COMPATIBILITY WITH OTHER ORACLES

// INFORMATIONAL

Description

The current implementation of the **AaveOracle** contract relies on **Pyth** as the primary source of asset prices and falls back to a secondary oracle if the **Pyth** price is not available or inconsistent. However, this design may limit the flexibility and extensibility of the oracle system in the future, as it is tightly coupled with **Pyth** and the fallback oracle.

- contracts/misc/AaveOracle.sol [Lines: 101-114]

```
101   /// @inheritdoc IPriceOracleGetter
102   function getAssetPrice(address asset) public view override returns (uin
103     bytes32 priceFeedId = priceFeedIds[asset];
104
105     if (asset == BASE_CURRENCY) return BASE_CURRENCY_UNIT;
106
107     if (priceFeedId == 0) return 0;
108
109     PythStructs.Price memory priceStruct = IPyth(ADDRESSES_PROVIDER.getPy
110       priceFeedId
111     );
112
113     return uint256(uint64(priceStruct.price));
114   }
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to introduce a more generic interface for price sources, allowing the integration of various oracle solutions beyond Chainlink and Pyth.

Remediation

ACKNOWLEDGED: The **Colend team** acknowledged the issue.

7.4 DISABLE BORROWING ON THE STABLEDEBT TOKENS

// INFORMATIONAL

Description

On November 4th, 2023, Aave received a report through their bug bounty program about a high severity vulnerability affecting Aave v2, which was later upgraded to a critical severity. The vulnerability impacted some assets in Aave v2 on Ethereum and Aave v3 on Optimism, Arbitrum, Avalanche, and Polygon.

However, it has been discovered that the Colend protocol, did not disable the stable debt token minting functionality in their implementation. This leaves the Colend protocol potentially vulnerable to the same security issue that affected Aave.

- contracts/protocol/tokenization/StableDebtToken.sol [Lines: 123-175]

```
123   /// @inheritdoc IStableDebtToken
124   function mint(
125     address user,
126     address onBehalfOf,
127     uint256 amount,
128     uint256 rate
129   ) external virtual override onlyPool returns (bool, uint256, uint256) {
130     MintLocalVars memory vars;
131
132     if (user != onBehalfOf) {
133       _decreaseBorrowAllowance(onBehalfOf, user, amount);
134     }
135
136     (, uint256 currentBalance, uint256 balanceIncrease) = _calculateBalan
137
138     vars.previousSupply = totalSupply();
139     vars.currentAvgStableRate = _avgStableRate;
140     vars.nextSupply = _totalSupply = vars.previousSupply + amount;
141
142     vars.amountInRay = amount.wadToRay();
143
144     vars.currentStableRate = _userState[onBehalfOf].additionalData;
145     vars.nextStableRate = (vars.currentStableRate.rayMul(currentBalance.w
146       vars.amountInRay.rayMul(rate)).rayDiv((currentBalance + amount).wad
147
148     _userState[onBehalfOf].additionalData = vars.nextStableRate.toUint128
149
150     //solium-disable-next-line
```

```
151
152     _totalSupplyTimestamp = _timestamps[onBehalfOf] = uint40(block.timest
153
154     // Calculates the updated average stable rate
155     vars.currentAvgStableRate = _avgStableRate = (
156       (vars.currentAvgStableRate.rayMul(vars.previousSupply.wadToRay()) +
157         rate.rayMul(vars.amountInRay)).rayDiv(vars.nextSupply.wadToRay())
158     ).toUint128();
159
160     uint256 amountToMint = amount + balanceIncrease;
161     _mint(onBehalfOf, amountToMint, vars.previousSupply);
162
163     emit Transfer(address(0), onBehalfOf, amountToMint);
164     emit Mint(
165       user,
166       onBehalfOf,
167       amountToMint,
168       currentBalance,
169       balanceIncrease,
170       vars.nextStableRate,
171       vars.currentAvgStableRate,
172       vars.nextSupply
173     );
174
175     return (currentBalance == 0, vars.nextSupply, vars.currentAvgStableRa
   }
```

Score

Impact:

Likelihood:

Recommendation

To mitigate the risk and ensure the security of the Colend protocol, it is strongly recommended to disable the stable debt token minting functionality.

Remediation

ACKNOWLEDGED: The **Colend team** acknowledged the issue.

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.

```
INFO:Detectors:
AaveOracle.constructor(IPoolAddressesProvider,address[],bytes32[],address,address,uint256).baseCurrency (contracts/misc/AaveOracle.sol#53) lacks a zero-check on :
                - BASE_CURRENCY = baseCurrency (contracts/misc/AaveOracle.sol#59)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
AaveOracle.getAssetPrice(address) (contracts/misc/AaveOracle.sol#102-114) has external calls inside a loop: priceStruct = IPyth(ADDRESSES_PROVIDER.getPyth()).getPrice(priceFeedId) (contracts/misc/AaveOracle.sol#109-111)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Different versions of Solidity are used:
        - Version used: ['^0.8.0', '^0.8.10']
        - ^0.8.0 (contracts/dependencies/pyth/IPyth.sol#2)
        - ^0.8.0 (contracts/dependencies/pyth/IPythEvents.sol#2)
        - ^0.8.0 (contracts/dependencies/pyth/PythStructs.sol#2)
        - ^0.8.0 (contracts/interfaces/IACLManager.sol#2)
        - ^0.8.0 (contracts/interfaces/IAaveOracle.sol#2)
        - ^0.8.0 (contracts/interfaces/IPoolAddressesProvider.sol#2)
        - ^0.8.0 (contracts/interfaces/IPriceOracleGetter.sol#2)
        - ^0.8.0 (contracts/protocol/libraries/helpers/Errors.sol#2)
        - ^0.8.10 (contracts/misc/AaveOracle.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Pragma version^0.8.0 (contracts/dependencies/pyth/IPyth.sol#2) allows old versions
Pragma version^0.8.0 (contracts/dependencies/pyth/IPythEvents.sol#2) allows old versions
Pragma version^0.8.0 (contracts/dependencies/pyth/PythStructs.sol#2) allows old versions
Pragma version^0.8.0 (contracts/interfaces/IACLManager.sol#2) allows old versions
Pragma version^0.8.0 (contracts/interfaces/IAaveOracle.sol#2) allows old versions
Pragma version^0.8.0 (contracts/interfaces/IPoolAddressesProvider.sol#2) allows old versions
Pragma version^0.8.0 (contracts/interfaces/IPriceOracleGetter.sol#2) allows old versions
Pragma version^0.8.10 (contracts/misc/AaveOracle.sol#2) allows old versions
Pragma version^0.8.0 (contracts/protocol/libraries/helpers/Errors.sol#2) allows old versions
solc-0.8.19 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Function IACLManager.ADDRESSES_PROVIDER() (contracts/interfaces/IACLManager.sol#16) is not in mixedCase
Function IACLManager.POOL_ADMIN_ROLE() (contracts/interfaces/IACLManager.sol#22) is not in mixedCase
Function IACLManager.EMERGENCY_ADMIN_ROLE() (contracts/interfaces/IACLManager.sol#28) is not in mixedCase
Function IACLManager.RISK_ADMIN_ROLE() (contracts/interfaces/IACLManager.sol#34) is not in mixedCase
Function IACLManager.FLASH_BORROWER_ROLE() (contracts/interfaces/IACLManager.sol#40) is not in mixedCase
Function IACLManager.BRIDGE_ROLE() (contracts/interfaces/IACLManager.sol#46) is not in mixedCase
Function IACLManager.ASSET_LISTING_ADMIN_ROLE() (contracts/interfaces/IACLManager.sol#52) is not in mixedCase
Function IAaveOracle.ADDRESSES_PROVIDER() (contracts/interfaces/IAaveOracle.sol#37) is not in mixedCase
Function IPriceOracleGetter.BASE_CURRENCY() (contracts/interfaces/IPriceOracleGetter.sol#15) is not in mixedCase
Function IPriceOracleGetter.BASE_CURRENCY_UNIT() (contracts/interfaces/IPriceOracleGetter.sol#22) is not in mixedCase
Variable AaveOracle.ADDRESSES_PROVIDER (contracts/misc/AaveOracle.sol#21) is not in mixedCase
Variable AaveOracle.BASE_CURRENCY (contracts/misc/AaveOracle.sol#27) is not in mixedCase
Variable AaveOracle.BASE_CURRENCY_UNIT (contracts/misc/AaveOracle.sol#28) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

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

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.