// HALBORN

ORAICHAIN aiVaultORAI Token

Smart Contract Security Audit

Prepared by: **Halborn**
Date of Engagement: November 27, 2020
Visit: Halborn.com

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	11/27/2020	Gabi Urrutia
0.2	Document Edits	11/30/2020	Gabi Urrutia
1.0	Final Version	12/01/2020	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

1.1 INTRODUCTION

The Oraichain engaged Halborn to conduct a security assessment on their aiVaultORAI Token smart contract beginning on November 24th, 2020 and

ending November 30th, 2020. The security assessment was scoped to the contract aiVaultORAI.sol and an audit of the security risk and implications regarding the changes introduced by the development team at Oraichain prior to its production release shortly following the assessments deadline.

Overall, the smart contracts code is extremely well documented, follows a high-quality software development standard, contain many utilities and automation scripts to support continuous deployment / testing / integration, and does NOT contain any obvious exploitation vectors that Halborn was able to leverage within the timeframe of testing allotted.

Though the outcome of this security audit is satisfactory; due to time and resource constraints, only testing and verification of essential properties related to the Token Contract was performed to achieve objectives and deliverables set in the scope. It is important to remark the use of the best practices for secure smart contract development.

Halborn recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.2 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. 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 audit:

- Research into architecture, purpose, and use of Governance Token.
- Smart Contract manual code read 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.
- Scanning of solidity files for vulnerabilities, security hotspots, or bugs. (MythX)
- Static Analysis of security for scoped contract and imported functions. (Slither)
- Smart Contract analysis and automatic exploitation (limited-time)
- Symbolic Execution / EVM bytecode security assessment (limited-time)

1.3 SCOPE

IN-SCOPE:
Code related to the aiVaultORAI smart contract. Specific commit of contract: commit 849a1ed96bf3699d55d09af9cfe0a0350609a0dd

OUT-OF-SCOPE:
External contracts, External Oracles, other smart contracts in the repository or imported by aiVaultORAI, economic attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	3

SECURITY ANALYSIS	RISK LEVEL
FLOATING PRAGMA	Low
DIVIDE BEFORE MULTIPLY	Low
STRICT EQUALITY	Very Low
FOR LOOP OVER DYNAMIC ARRAY	Informational
USE OF INLINE ASSEMBLY	Informational
NO RETURN VALUE	Informational
STATIC ANALYSIS REPORT	Informational
ERC CONFORMAL CHECKER	Informational
AUTOMATED SECURITY SCAN	Informational

FINDINGS & TECH DETAILS

3.1 FLOATING PRAGMA - LOW

Description

Oraichain contract use the floating pragma ^0.5.16. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. At the time of this audit, the current version is already at 0.7 The newer versions provide features that provide checks and accounting, as well as prevent insecure use of code.

Code Location

aiVaultORAI.sol Line #5

```
1  /**
2   *Submitted for verification at Etherscan.io on 2020-10-14
3  */
4
5  pragma solidity ^0.5.16;
6
```

Recommendation

Consider lock the pragma version known bugs for the compiler version. When possible, do not use floating pragma in the final live deployment.

3.2 DIVIDE BEFORE MULTIPLY - LOW

Description:

Solidity integer division might truncate. As a result, performing multiplication before division might reduce precision. Due to the sensitivity of precision, and the amount of detail the development team is putting on the dynamic balancing mechanics involved in Oraichain, this may be a factor in accuracy of weights/rates. In this case, the parenthesis seem to be unnecessary.

Code Location:

aiVaultORAI.sol Line #562-563

```
560     function checkReward(address _reciver) public view returns (uint256) {
561         uint256 _balance = balanceOf(_reciver);
562         uint256 amount = (_balance.mul(rewardAmount)).div(totalSupply()).mul(
563             10**18
564         );
565         return amount;
566     }
```

Recommendation:

Consider ordering multiplication before division.

uint256 amount = _balance.mul(rewardAmount).div(totalSupply()).mul(10**18). Therefore, it's possible to move the mul(10**18) to the beginning of the expression.

3.3 STRICT EQUALITY - VERY LOW

Description:

Use of strict equalities that can be easily manipulated by an attacker.

Code Locatioin:

aiVaultORAI.sol Line #548

```
548         if (totalSupply() == 0) {
549             shares = _amount;
550         } else {
551             shares = (_amount.mul(totalSupply())).div(_pool);
```

Recommendation:

While these sections of code use it for time, and weight adjustments, do not use strict equality to determine if an account has enough Ether or tokens

3.4 FOR LOOP OVER DYNAMIC ARRAY - INFORMATIONAL

Description:

Calls inside a loop might lead to a denial-of-service attack. The

function discovered is a for loop on variable `i` that iterates up to the addressList variable. If this integer is evaluated at extremely large numbers, or `i` is reset by external calling functions, this can cause a DoS. An attack could register a huge amount of address, causing the problem described above.

Code Location:

aiVaultORAI.sol Line #575-585

```
573     function reward(address[] memory _addressList) public {
574         // require(msg.sender == governance, "!governance");
575         for (uint256 i = 0; i < _addressList.length; i++) {
576             address _address = _addressList[i];
577             uint256 _balance = balanceOf(_address);
578             if (_balance > 0) {
579                 uint256 amount = (_balance.mul(10**18).mul(rewardAmount)).div(
580                     totalSupply()
581                 );
582                 ERC20(rewardAddress).transferFrom(msg.sender, _address, amount);
583             }
584         }
585     }
```

Recommendation:

If possible, avoid actions that require looping across the entire data structure.

3.5 USE OF INLINE ASSEMBLY - INFORMATIONAL

Description:

Inline assembly is a way to access the Etehreum Virtual Machine at a low level. This discards several important safety features in Solidity.

Code Location:

aiVaultORAI.sol Line #332

```
331         // solhint-disable-next-line no-inline-assembly
332         assembly {
333             codehash := extcodehash(account)
334         }
335         return (codehash != 0x0 && codehash != accountHash);
336     }
```

Recommendation:

When possible, do not use inline assembly because it is a manner to access to the EVM (Ethereum Virtual Machine) at a low level. An attacker

could bypass many important safety features of Solidity.

3.6 NO RETURN VALUE - INFORMATIONAL

Description:

Defining a return type but the function is not explicitly returning any value.

Code Location:

aiVaultORAI.sol Line #573-585

```
573     function reward(address[] memory _addressList) public {
574         // require(msg.sender == governance, "!governance");
575         for (uint256 i = 0; i < _addressList.length; i++) {
576             address _address = _addressList[i];
577             uint256 _balance = balanceOf(_address);
578             if (_balance > 0) {
579                 uint256 amount = (_balance.mul(10**18).mul(rewardAmount)).div(
580                     totalSupply()
581                 );
582                 ERC20(rewardAddress).transferFrom(msg.sender, _address, amount);
583             }
584         }
585     }
```

Recommendation:

It will fail the execution of external caller if caller is expecting return value from these methods. To remediate the issue, remove unnecessary return value type if method is not intended to return any value.

3.7 STATIC ANALYSIS REPORT - INFORMATIONAL

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the Governance Token contract. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of

the contracts' APIs across the entire codebase.

Results:

```
ziion@ziion-sec554:~/oraichain$ slither aiVaultORAI.sol
INFO:Detectors:
aiORAI.checkReward(address) (aiVaultORAI.sol#564-570) performs a multiplication on the result of a division:
        -amount = (_balance.mul(rewardAmount)).div(totalSupply()).mul(10 ** 18) (aiVaultORAI.sol#566-568)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
aiORAI.deposit(uint256) (aiVaultORAI.sol#545-558) uses a dangerous strict equality:
        - totalSupply() == 0 (aiVaultORAI.sol#552)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
aiORAI.reward(address[]) (aiVaultORAI.sol#577-589) ignores return value by ERC20(rewardAddress).transferFrom(msg.sender,_address,amount) (aiVaultORAI.sol#586)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

```
INFO:Detectors:
ERC20Detailed.constructor(string,string,uint8).name (aiVaultORAI.sol#242) shadows:
        - ERC20Detailed.name() (aiVaultORAI.sol#251-253) (function)
ERC20Detailed.constructor(string,string,uint8).symbol (aiVaultORAI.sol#243) shadows:
        - ERC20Detailed.symbol() (aiVaultORAI.sol#255-257) (function)
ERC20Detailed.constructor(string,string,uint8).decimals (aiVaultORAI.sol#244) shadows:
        - ERC20Detailed.decimals() (aiVaultORAI.sol#259-261) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
aiORAI.reward(address[]) (aiVaultORAI.sol#577-589) has external calls inside a loop: ERC20(rewardAddress).transferFrom(msg.sender,_address,amount) (aiVaultORAI.sol#586)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Reentrancy in aiORAI.deposit(uint256) (aiVaultORAI.sol#545-558):
        External calls:
        - token.safeTransferFrom(msg.sender,address(this),_amount) (aiVaultORAI.sol#548)
        State variables written after the call(s):
        - _mint(msg.sender,shares) (aiVaultORAI.sol#557)
                - _balances[account] = _balances[account].add(amount) (aiVaultORAI.sol#196)
        - _mint(msg.sender,shares) (aiVaultORAI.sol#557)
                - _totalSupply = _totalSupply.add(amount) (aiVaultORAI.sol#195)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in aiORAI.deposit(uint256) (aiVaultORAI.sol#545-558):
        External calls:
        - token.safeTransferFrom(msg.sender,address(this),_amount) (aiVaultORAI.sol#548)
        Event emitted after the call(s):
        - Transfer(address(0),account,amount) (aiVaultORAI.sol#197)
                - _mint(msg.sender,shares) (aiVaultORAI.sol#557)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Address.isContract(address) (aiVaultORAI.sol#329-340) uses assembly
        - INLINE ASM (aiVaultORAI.sol#336-338)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (aiVaultORAI.sol#350-362):
        - (success) = recipient.call.value(amount)() (aiVaultORAI.sol#357)
Low level call in SafeERC20.callOptionalReturn(IERC20,bytes) (aiVaultORAI.sol#444-459):
        - (success,returndata) = address(token).call(data) (aiVaultORAI.sol#448)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
```

3.7.1 ERC CONFORMAL CHECKER - INFORMATIONAL

Description:

Another Slither tool can test ERC Token functions. Thus, slither-check-erc20 was performed over aiVaultORAI:

```
# Check aiORAI

## Check functions
[✓] totalSupply() is present
        [✓] totalSupply() -> () (correct return value)
        [✓] totalSupply() is view
[✓] balanceOf(address) is present
        [✓] balanceOf(address) -> () (correct return value)
        [✓] balanceOf(address) is view
[✓] transfer(address,uint256) is present
        [✓] transfer(address,uint256) -> () (correct return value)
        [✓] Transfer(address,address,uint256) is emitted
[✓] transferFrom(address,address,uint256) is present
        [✓] transferFrom(address,address,uint256) -> () (correct return value)
        [✓] Transfer(address,address,uint256) is emitted
[✓] approve(address,uint256) is present
        [✓] approve(address,uint256) -> () (correct return value)
        [✓] Approval(address,address,uint256) is emitted
[✓] allowance(address,address) is present
        [✓] allowance(address,address) -> () (correct return value)
        [✓] allowance(address,address) is view
[✓] name() is present
        [✓] name() -> () (correct return value)
        [✓] name() is view
[✓] symbol() is present
        [✓] symbol() -> () (correct return value)
        [✓] symbol() is view
[✓] decimals() is present
        [✓] decimals() -> () (correct return value)
        [✓] decimals() is view

## Check events
[✓] Transfer(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
[✓] Approval(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed


        [✓] aiORAI has increaseAllowance(address,uint256)
```

Results:

All tests were successfully passed.

3.8 AUTOMATED SECURITY SCAN - INFORMATIONAL

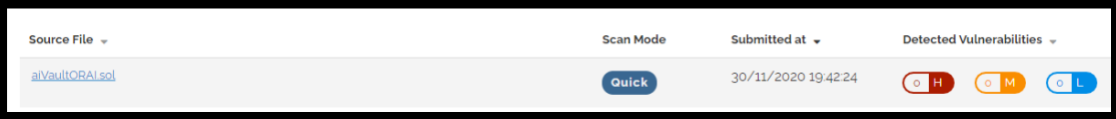
Description:

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the testers machine and sent the compiled results to the analyzers to locate any vulnerabilities. Security Detections are only in scope, and the analysis was pointed towards issues with aiVaultORAI.

Results

aiVaultORAI

MythX detected 0 High findings, 0 Medium, and 0 Low.



THANK YOU FOR CHOOSING

// HALBORN