// HALBORN

DRAFT

NewOrderDAO - Y2K Finance

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **July 29th, 2022 - August 4th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	08/04/2022	Pawel Bartunek
0.2	Document Updated	08/05/2022	Pawel Bartunek
0.3	Draft Review	08/05/2022	Kubilay Onur Gungor
0.4	Draft Review	08/05/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Kubilay Onur Gungor	Halborn	kubilay.gungor@halborn.com
Pawel Bartunek	Halborn	pawel.bartunek@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

NewOrderDAO engaged Halborn to conduct a security audit on their smart contracts beginning on July 29th, 2022 and ending on August 4th, 2022 . The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that need to be reviewed by the NewOrderDAO team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this audit. 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 security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart contract manual code review and walk through.
- 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 contract, and imported functions (Slither).
- Local or Testnet deployment (Brownie, Foundry, Remix IDE).

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.

1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The assessment was scoped to the repositories listed below:

- Y2K core, Rewards
  - Commit: **5dcfcf03b2c9a861679a810f1c5e2ebe0dcb21cc**
  - New contracts:
    - PegOracle.sol
    - RewardsFactory.sol
  - Modified contracts (withdraw fee implementation):
    - Vault.sol
    - VaultFactory.sol

Out-of-Scope: Other smart contracts in the repository, external libraries and economical attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	2	3

LIKELIHOOD

IMPACT

(HAL-02)				
	(HAL-01)			
	(HAL-03) (HAL-04)			
(HAL-05) (HAL-06) (HAL-07)				

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 WITHDRAW FEE IS NOT VALIDATED	Medium	-
HAL-02 PEGORACLE CONTRACT HAS HARD CODED DECIMALS	Medium	-
HAL-03 DIVIDE BEFORE MULTIPLY	Low	-
HAL-04 PEGORACLE CONTRACT IS NOT CHECKING THE SEQUENCER STATE	Low	-
HAL-05 MISSING ZERO ADDRESS CHECK	Informational	-
HAL-06 CREATESTAKINGREWARDS COULD BE DEFINED AS EXTERNAL	Informational	-
HAL-07 MISSING NATSPEC DOCUMENTATION	Informational	-

FINDINGS & TECH DETAILS

3.1 (HAL-01) WITHDRAW FEE IS NOT VALIDATED - MEDIUM

Description:

The withdrawalFee parameter of the Vault can be set to over 100% using changeWithdrawalFee function. Such a high fee will cause a transaction to revert because of an integer underflow error during a withdrawal.

Code Location:

Setter, not validating withdraw fee value:

Listing 1: Y2K-smartcontracts/Core Earthquake/src/Vault.sol (Line 344)

```
343 function changeWithdrawalFee(uint256 _riskWithdrawalFee) public
 ↳ onlyFactory {
344     withdrawalFee = _riskWithdrawalFee;
345 }
```

Part of the withdraw function, subtracting calculated fee value from entitled shares:

Listing 2: Y2K-smartcontracts/Core Earthquake/src/Vault.sol (Line 240)

```
238 //Taking fee from the amount
239 uint feeValue = calculateWithdrawalFeeValue(entitledShares);
240 entitledShares = entitledShares - feeValue;
241 asset.safeTransfer(treasury, feeValue);
```

Proof of Concept:

Foundry test case, changing withdraw fee to 120%, causing arithmetic error:

Listing 3: withdrawalFee underflow (Line 26)

```
1  function testWithdrawFeeOverflow() public {
2
3      uint256 fee = 10;
4      uint256 withdrawalFee = 50;
5      int256 strikePrice = 120000000; //1$ = 100000000
6      uint256 epochBegin = block.timestamp + 2 days;
7      uint256 epochEnd = block.timestamp  + 30 days;
8
9      (address insr, address risk) = CreationNewVaults(
10         fee,
11         withdrawalFee,
12         USDC,
13         strikePrice,
14         epochBegin,
15         epochEnd,
16         USDC_oracle,
17         "Y2K.USDC_1,20$"
18     );
19
20     address riskUser = address(3);
21
22     DepositRisk(riskUser, 20 ether, epochEnd, risk);
23     NODepegKeeper(epochEnd, insr, risk);
24
25     // update withrdaw fee to 120%
26     vaultFactory.changeWithdrawalVaultFee(1, 1200);
27
28     // withrdaw risk
29     vm.startPrank(riskUser);
30     Vault riskVault = Vault(risk);
31     uint256 risk_user_vaultbalance = riskVault.balanceOf(riskUser,
↳  epochEnd);
32
33     vm.expectRevert(stdError.arithmeticError);
34     riskVault.withdraw(
35         epochEnd,
36         risk_user_vaultbalance,
37         riskUser,
38         riskUser
39     );
40     vm.stopPrank();
41 }
```

Risk Level:

Likelihood - 2
Impact - 4

Recommendation:

It is recommended to add a boundary check for the fee value setter. It should not be possible to set fees over 100%.

3.2 (HAL-02) PEGORACLE CONTRACT HAS HARD-CODED DECIMALS - MEDIUM

Description:

The PegOracle.sol contract is using the hard-coded value of 10e7 in the price calculation. Some price feeds can use a different number of decimals.

Most of the Arbitrum price feeds are using 8 decimals, but there are feeds with 9 or 18.

Using feeds with a different number of decimals with the current implementation may cause improper rate calculation.

Code Location:

Listing 4: Y2K-smartcontracts/Core Earthquake/src/PegOracle.sol (Line 56)

```
56 return (roundID1, (price1*10e7/price2*10e7)/10e7, startedAt1,
↳ timeStamp1, answeredInRound1);
```

Proof of Concept:

Example price calculation in Brownie, simulating oracles with different a number of decimals (8 and 18):

Listing 5: rate calculation in Brownie (Lines 13,22)

```
1 oracle = Contract.from_explorer("0
↳ x07C5b924399cc23c24a95c8743DE4006a32b7f2a")
2 oracle2 = Contract.from_explorer("0
↳ x639Fe6ab55C921f74e7fac1ee960C0B6293ba612")
3 >>> price1 = oracle.latestRoundData()[1]
4 >>> price2 = oracle2.latestRoundData()[1]
5 >>> price1
```

```
161333957890
>>> price2
165018000000
>>> price2 = price2 * 10e9
>>> int(price2)
1650179999999999868928
>>> int((price1*10e7/price2*10e7)/10e7)
0
>>> price1 = oracle.latestRoundData()[1]
>>> price2 = oracle2.latestRoundData()[1]
>>> price1 = price1 * 10e9
>>> int(price1)
1613339578900000014336
>>> int(price2)
165018000000
>>> int((price1*10e7/price2*10e7)/10e7)
977674907525239808
```

Similar test case in Foundry:

Listing 6: Foundry PoC (Line 11)

```
1  function testOracleDifferentDecimals() public {
2      int256 price1 = stETH_Oracle.getOracle1_Price();
3      int256 price2 = stETH_Oracle.getOracle2_Price();
4
5      price2 = price2 * 10e9;
6
7      int256 rate = int256(price1*10e7/price2*10e7)/10e7;
8
9      emit log_named_int("Oracle 1 Price", price1);
10     emit log_named_int("Oracle 2 Price", price2);
11     emit log_named_int("Rate: ", rate);
12     assertEq(rate, 0);
13 }
```

Risk Level:

Likelihood - 1
Impact - 5

Recommendation:

OpenZeppelin guidelines regarding Chainlink price feeds recommends to: “Always check the units and decimals for each price feed.”.

It is recommended to consider using decimals() method to define a number of decimals for a given price feed and adjust calculation, or verify the number of Oracle decimals in the constructor and allow only ones with the desired number of decimals.

3.3 (HAL-03) DIVIDE BEFORE MULTIPLY - LOW

Description:

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision. In the PegOracle contract, the rate calculation formula the division is being performed before the multiplication operation.

Code Location:

Listing 7: Y2K-smartcontracts/Core Earthquake/src/PegOracle.sol (Line 56)

```
56 return (roundID1, (price1*10e7/price2*10e7)/10e7, startedAt1,
↳ timeStamp1, answeredInRound1);
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Consider doing multiplication operation before division to prevail precision in the values in non-floating data type.

3.4 (HAL-04) PEGORACLE CONTRACT IS NOT CHECKING THE SEQUENCER STATE - LOW

Description:

The PegOracle.sol contract is not validating the sequencer state.

According to Chainlink documentation, when consuming a price feed on Arbitrum, it is a good practice to check the sequencer state:

“As a best practice, use the L2 sequencer feed to verify the status of the sequencer when running applications on the Arbitrum network. See the L2 Sequencer Uptime Feeds page for examples.”
source: Chainlink documentation

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Consider adding a sequencer check to the price feed consumer implementation.
References:
Chainlink example code.
Arbitrum sequencer flag.

3.5 (HAL-05) MISSING ZERO ADDRESS CHECK - INFORMATIONAL

Description:

The constructors of PegOracle.sol contract is not validating oracle addresses. It is possible to set an oracle with 0x0 address in PegOracle contract, making getting a price impossible.

The constructor of RewardsFactory is not validating supplied addresses.

Code Location:

Listing 8: Y2K-smartcontracts/Core Earthquake/src/PegOracle.sol (Lines 15,16)

```
14 constructor(address _oracle1, address _oracle2) {
15     oracle1 = _oracle1;
16     oracle2 = _oracle2;
17 }
```

Listing 9: Y2K-smartcontracts/Core Earthquake/src/rewards/RewardsFactory.sol (Lines 23,24,25)

```
22 constructor(address _govToken, address _factory) {
23     admin = msg.sender;
24     govToken = _govToken;
25     factory = _factory;
26 }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider adding a proper address validation in every state variable assignment done from user-supplied input.

3.6 (HAL-06) CREATESTAKINGREWARDS COULD BE DEFINED AS EXTERNAL - INFORMATIONAL

Description:

Public functions that are never called by the contract should be declared external to save gas.

Code Location:

- createStakingRewards function in RewardsFactory.sol contract
- latestRoundData, getOracle1_Price, getOracle2_Price functions in PegOracle.sol contract

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Use the external attribute for functions never called from the contract.

3.7 (HAL-07) MISSING NATSPEC DOCUMENTATION - INFORMATIONAL

Description:

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

Code Location:

contracts:

- PegOracle.sol
- RewardsFactory.sol

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider adding NatSpec documentation at the beginning of each function.

AUTOMATED TESTING

4.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. 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 all-scoped 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.

Slither results:

PegOracle.sol:

Listing 10

```
PegOracle.latestRoundData() (src/PegOracle.sol#19-57) performs a multiplication on the result of a division:
        -(roundID1,(price1 * 10e7 / price2 * 10e7) / 10e7,startedAt1,timeStamp1,answeredInRound1) (src/PegOracle.sol#56)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

PegOracle.constructor(address,address)._oracle1 (src/PegOracle.sol#14) lacks a zero-check on :
                - oracle1 = _oracle1 (src/PegOracle.sol#15)
PegOracle.constructor(address,address)._oracle2 (src/PegOracle.sol#14) lacks a zero-check on :
                - oracle2 = _oracle2 (src/PegOracle.sol#16)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Different versions of Solidity are used:
        - Version used: ['0.8.15', '^0.8.0']
        - ^0.8.0 (lib/chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol#2)
        - 0.8.15 (src/PegOracle.sol#2)
```

```
15 Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#different-pragma-directives-are-used
16
17 Pragma version^0.8.0 (lib/chainlink/contracts/src/v0.8/interfaces/
↳ AggregatorV3Interface.sol#2) allows old versions
18 Pragma version0.8.15 (src/PegOracle.sol#2) necessitates a version
↳ too recent to be trusted. Consider deploying with
↳ 0.6.12/0.7.6/0.8.7
19 solc-0.8.15 is not recommended for deployment
20 Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#incorrect-versions-of-solidity
21
22 Function PegOracle.getOracle1_Price() (src/PegOracle.sol#59-76) is
↳  not in mixedCase
23 Function PegOracle.getOracle2_Price() (src/PegOracle.sol#78-95) is
↳  not in mixedCase
24 Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#conformance-to-solidity-naming-conventions
25
26 Variable PegOracle.getOracle1_Price().answeredInRound1 (src/
↳ PegOracle.sol#68) is too similar to PegOracle.latestRoundData().
↳ answeredInRound2 (src/PegOracle.sol#47)
27 Variable PegOracle.latestRoundData().answeredInRound1 (src/
↳ PegOracle.sol#32) is too similar to PegOracle.latestRoundData().
↳ answeredInRound2 (src/PegOracle.sol#47)
28 Variable PegOracle.getOracle2_Price().answeredInRound1 (src/
↳ PegOracle.sol#87) is too similar to PegOracle.latestRoundData().
↳ answeredInRound2 (src/PegOracle.sol#47)
29 Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#variable-names-are-too-similar
30
31 latestRoundData() should be declared external:
32         - PegOracle.latestRoundData() (src/PegOracle.sol#19-57)
33 getOracle1_Price() should be declared external:
34         - PegOracle.getOracle1_Price() (src/PegOracle.sol#59-76)
35 getOracle2_Price() should be declared external:
36         - PegOracle.getOracle2_Price() (src/PegOracle.sol#78-95)
37 Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#public-function-that-could-be-declared-external
38 src/PegOracle.sol analyzed (2 contracts with 78 detectors), 15
↳ result(s) found
```

RewardsFactory.sol

Listing 11

```
RewardsFactory.constructor(address,address)._factory (src/rewards/
↳ RewardsFactory.sol#22) lacks a zero-check on :
                - factory = _factory (src/rewards/RewardsFactory.
↳ sol#25)
Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#missing-zero-address-validation
Parameter RewardsFactory.createStakingRewards(uint256,uint256).
↳ _marketIndex (src/rewards/RewardsFactory.sol#44) is not in
↳ mixedCase
Parameter RewardsFactory.createStakingRewards(uint256,uint256).
↳ _epoch (src/rewards/RewardsFactory.sol#44) is not in mixedCase
Parameter RewardsFactory.getHashedIndex(uint256,uint256)._index (
↳ src/rewards/RewardsFactory.sol#62) is not in mixedCase
Parameter RewardsFactory.getHashedIndex(uint256,uint256)._epoch (
↳ src/rewards/RewardsFactory.sol#62) is not in mixedCase
Variable RewardsFactory.hashedIndex_StakingRewards (src/rewards/
↳ RewardsFactory.sol#38) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#conformance-to-solidity-naming-conventions
createStakingRewards(uint256,uint256) should be declared external:
        - RewardsFactory.createStakingRewards(uint256,uint256) (
↳ src/rewards/RewardsFactory.sol#44-60)
Reference: https://github.com/crytic/slither/wiki/Detector-
↳ Documentation#public-function-that-could-be-declared-external
```

The Slither results were analyzed, confirmed findings are included in the report. Remaining issues are connected to external libraries or best practices (like similar variable names, mixed case, etc).

THANK YOU FOR CHOOSING

// HALBORN