FINANCE.VOTE
Liquidity Mining
Smart Contract Security Audit

Prepared by: **Halborn**
Date of Engagement: January 4-11, 2021
Visit: Halborn.com

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	1/5/2021	Gabi Urrutia
0.2	Document Edits	1/8/2021	Gabi Urrutia
1.0	Final Version	1/11/2021	Gabi Urrutia

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

Finance.Vote engaged Halborn to conduct a security assessment on their LiquidityMining smart contracts beginning on January 4th, 2021 and ending January 11st, 2021. The security assessment was scoped to the contract LiquidityMining and an audit of the security risk and implications regarding the changes introduced by the development team at Finance.Vote prior to its production release shortly following the assessments deadline.

The most important security finding was a possible re-entrancy vulnerability in in updateSlot function due to the order of the calls into functions. This vulnerability was immediately fixed by Finance.Vote Team and tested it again by Halborn auditors.

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 Liquidity 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 LiquidityMining.
- 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 LiquidityMining smart contract.
Specific commit of contract: commit
f608d09702f864eda144613ebf4468b5a6783689

OUT-OF-SCOPE:
Other smart contracts in the repository and economics attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

SECURITY ANALYSIS	RISK LEVEL	Remediation Date
RE-ENTRANCY	Medium	1/11/2021
PRAGMA VERSION	Informational	-
POSSIBLE MISUSE OF PUBLIC FUNCTIONS	Informational	-
STATIC ANALYSIS	Medium	1/11/2021
AUTOMATED SECURITY SCAN RESULTS	Informational	-

FINDINGS & TECH DETAILS

3.1 RE-ENTRANCY - MEDIUM

Description:

Calling external contracts is dangerous if some functions and variables are called after the external call. An attacker could use a malicious contract to perform a recursive call before calling function and take over the control flow. transfer function is executed without check the totalRewards value before. Thus, an attacker could perform a recursive call to execute malicious code.

Code Location:

LiquidityMining.sol Line #162-182

```
162     function updateSlot(uint slotId) public {
163         Slot storage slot = slots[slotId];
164 
165         // burn and rewards always have to update together, since they both depend on lastUpdatedBlock
166         uint burn = getBurn(slotId);
167         if (burn > 0) {
168             liquidityToken.transfer(address(0), burn);
169             slot.deposit = slot.deposit.minus(burn);
170         }
171 
172         uint rewards = getRewards(slotId);
173         if (rewards > 0) {
174             baseToken.transfer(slot.owner, rewards);
175 
176             // bookkeeping
177             totalRewards = totalRewards.plus(rewards);
178             totalRewardsFor[slot.owner] = totalStakedFor[slot.owner].plus(rewards);
179         }
180 
181         slot.lastUpdatedBlock = block.number;
182     }
```

Recommendation:

As possible, external calls should be at the end of the function in order to avoiding an attacker take over the control flow. In that case, check totalRewards before call transfer function. totalRewards and totalRewardsFor variables should be specifically call before call liquidityToken.transfer(address(0), rewards/burn) and baseToken.transfer(slot.owner, rewards/burn);

```
        uint burn = getBurn(slotId);
        if (burn > 0) {
            totalRewards = totalRewards.plus(burn);
            totalRewardsFor[slot.owner] = totalStakedFor[slot.owner].plus(burn);

            liquidityToken.transfer(address(0), burn);
            slot.deposit = slot.deposit.minus(burn);
        }

        uint rewards = getRewards(slotId);
        if (rewards > 0) {

            totalRewards = totalRewards.plus(rewards);
            totalRewardsFor[slot.owner] = totalStakedFor[slot.owner].plus(rewards);

            baseToken.transfer(slot.owner, rewards);

            // bookkeeping

        }
```

3.2 PRAGMA VERSION - INFORMATIONAL

Description:

LiquidityMining contract uses one of the latest pragma version (0.7.4) which was released on October 19, 2020. The latest pragma version (0.8.0) was released in December 2020. Many pragma versions have been lately released, going from version 0.6.x to the recently released version 0.8.x. in just 6 months.

Reference: https://github.com/ethereum/solidity/releases

In the Solitidy Github repository, there is a json file where are all bugs finding in the different compiler versions. No bugs have been found in > 0.7.3 versions but very few in 0.7.0 - 0.7.3. So, the latest tested and stable version is pragma 0.6.12. Furthermore, pragma 0.612 is widely used by Solidity developers and has been extensively tested in many security audits.

Reference: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json

Code Location:

LiquidityMining.sol Line #3

```
1 // SPDX-License-Identifier: GPL-3.0-only
2
3 pragma solidity 0.7.4;
4
5 import "./SafeMathLib.sol";
6 import "./Token.sol";
```

Recommendation:

Consider if possible, using the latest stable pragma version that have been well tested to prevent potential undiscovered vulnerabilities.

3.3 POSSIBLE MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

In public functions, array arguments are immediately copied array to memory, while external functions can read directly from calldata. Reading calldata is cheaper than memory allocation. Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, function expects its arguments being in memory when the compiler generates the code for an internal function. In Liquidity Mining contract, many functions are never directly called by another function in the same contract.

Code Location:

LiquidityMining.sol Line #109-159

```
109     function setManagement(address newMgmt) public managementOnly {
110         address oldMgmt = management;
111         management = newMgmt;
112         emit ManagementUpdated(oldMgmt, newMgmt);
113     }
```

```
109     function setManagement(address newMgmt) public managementOnly {
110         address oldMgmt = management;
111         management = newMgmt;
112         emit ManagementUpdated(oldMgmt, newMgmt);
113     }
```

```
116     function setMaxStakers(uint newMaxStakers) public managementOnly {
117         uint oldMaxStakers = maxStakers;
118         maxStakers = newMaxStakers;
119         emit MaxStakersUpdated(oldMaxStakers, maxStakers);
120     }
```

```
123     function setMinDeposit(uint newMinDeposit) public managementOnly {
124         uint oldMinDeposit = minimumDeposit;
125         minimumDeposit = newMinDeposit;
126         emit MinDepositUpdated(oldMinDeposit, newMinDeposit);
127     }
```

```
130     function setMaxDeposit(uint newMaxDeposit) public managementOnly {
131         uint oldMaxDeposit = maximumDeposit;
132         maximumDeposit = newMaxDeposit;
133         emit MaxDepositUpdated(oldMaxDeposit, newMaxDeposit);
134     }
```

```
137     function setMinBurnRate(uint newMinBurnRate) public managementOnly {
138         uint oldMinBurnRate = minimumBurnRate;
139         minimumBurnRate = newMinBurnRate;
140         emit MinBurnRateUpdated(oldMinBurnRate, newMinBurnRate);
141     }
```

```
    function setPulseWavelength(uint newWavelength) public managementOnly {
        uint oldWavelength = pulseWavelengthBlocks;
        pulseWavelengthBlocks = newWavelength;
        pulseConstant = pulseAmplitudeFVT / pulseWavelengthBlocks.times(pulseWavelengthBlocks);
        pulseIntegral = pulseSum(newWavelength);
        emit WavelengthUpdated(oldWavelength, newWavelength);
    }
```

```
    function setPulseAmplitude(uint newAmplitude) public managementOnly {
        uint oldAmplitude = pulseAmplitudeFVT;
        pulseAmplitudeFVT = newAmplitude;
        pulseConstant = pulseAmplitudeFVT / pulseWavelengthBlocks.times(pulseWavelengthBlocks);
        pulseIntegral = pulseSum(pulseWavelengthBlocks);
        emit AmplitudeUpdated(oldAmplitude, newAmplitude);
    }
```

Recommendation:

Consider as much as possible declaring external instead of public variables. As for best practices, you should use external if you expect that the function will only ever be called externally and use public if you need to call the function internally. In that case, the functions are not called by another function in the same contract, so marking them as external could save gas.

3.4 STATIC ANALYSIS REPORT - MEDIUM

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 Liquidity Mining 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:

LiquidityMining.sol

```
Reentrancy in LiquidityMining.claimSlot(uint256,uint256,uint256) (contracts/LiquidityMining.sol#184-226):
        External calls:
        - updateSlot(slotId) (contracts/LiquidityMining.sol#200)
                - liquidityToken.transfer(address(0),burn) (contracts/LiquidityMining.sol#168)
                - baseToken.transfer(slot.owner,rewards) (contracts/LiquidityMining.sol#174)
        State variables written after the call(s):
        - totalStakedFor[slot.owner] = totalStakedFor[slot.owner].minus(slot.deposit) (contracts/LiquidityMining.sol#207)
Reentrancy in LiquidityMining.claimSlot(uint256,uint256,uint256) (contracts/LiquidityMining.sol#184-226):
        External calls:
        - updateSlot(slotId) (contracts/LiquidityMining.sol#200)
                - liquidityToken.transfer(address(0),burn) (contracts/LiquidityMining.sol#168)
                - baseToken.transfer(slot.owner,rewards) (contracts/LiquidityMining.sol#174)
        - withdrawFromSlotInternal(slotId) (contracts/LiquidityMining.sol#210)
                - liquidityToken.transfer(slot.owner,slot.deposit) (contracts/LiquidityMining.sol#246)
        State variables written after the call(s):
        - withdrawFromSlotInternal(slotId) (contracts/LiquidityMining.sol#210)
                - slot.deposit = 0 (contracts/LiquidityMining.sol#247)
        - slot.owner = msg.sender (contracts/LiquidityMining.sol#214)
        - slot.burnRate = newBurnRate (contracts/LiquidityMining.sol#215)
        - slot.deposit = deposit (contracts/LiquidityMining.sol#216)
        - totalStaked = totalStaked.plus(deposit) (contracts/LiquidityMining.sol#219)
        - totalStakedFor[msg.sender] = totalStakedFor[msg.sender].plus(deposit) (contracts/LiquidityMining.sol#220)
Reentrancy in LiquidityMining.updateSlot(uint256) (contracts/LiquidityMining.sol#162-182):
        External calls:
        - liquidityToken.transfer(address(0),burn) (contracts/LiquidityMining.sol#168)
        State variables written after the call(s):
        - slot.deposit = slot.deposit.minus(burn) (contracts/LiquidityMining.sol#169)
Reentrancy in LiquidityMining.updateSlot(uint256) (contracts/LiquidityMining.sol#162-182):
        External calls:
        - liquidityToken.transfer(address(0),burn) (contracts/LiquidityMining.sol#168)
        - baseToken.transfer(slot.owner,rewards) (contracts/LiquidityMining.sol#174)
        State variables written after the call(s):
        - slot.lastUpdatedBlock = block.number (contracts/LiquidityMining.sol#181)
Reentrancy in LiquidityMining.withdrawFromSlot(uint256) (contracts/LiquidityMining.sol#228-240):
        External calls:
        - updateSlot(slotId) (contracts/LiquidityMining.sol#230)
                - liquidityToken.transfer(address(0),burn) (contracts/LiquidityMining.sol#168)
                - baseToken.transfer(slot.owner,rewards) (contracts/LiquidityMining.sol#174)
        - withdrawFromSlotInternal(slotId) (contracts/LiquidityMining.sol#232)
                - liquidityToken.transfer(slot.owner,slot.deposit) (contracts/LiquidityMining.sol#246)
        State variables written after the call(s):
        - withdrawFromSlotInternal(slotId) (contracts/LiquidityMining.sol#232)
                - slot.deposit = 0 (contracts/LiquidityMining.sol#247)
        - slot.owner = address(0) (contracts/LiquidityMining.sol#235)
        - slot.lastUpdatedBlock = block.number (contracts/LiquidityMining.sol#236)
        - slot.burnRate = 0 (contracts/LiquidityMining.sol#237)
Reentrancy in LiquidityMining.withdrawFromSlotInternal(uint256) (contracts/LiquidityMining.sol#242-249):
        External calls:
        - liquidityToken.transfer(slot.owner,slot.deposit) (contracts/LiquidityMining.sol#246)
        State variables written after the call(s):
        - slot.deposit = 0 (contracts/LiquidityMining.sol#247)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
LiquidityMining.updateSlot(uint256) (contracts/LiquidityMining.sol#162-182) ignores return value by liquidityToken.transfer(address(0),burn) (contracts/LiquidityMining.sol#168)
LiquidityMining.updateSlot(uint256) (contracts/LiquidityMining.sol#162-182) ignores return value by baseToken.transfer(slot.owner,rewards) (contracts/LiquidityMining.sol#174)
LiquidityMining.claimSlot(uint256,uint256,uint256) (contracts/LiquidityMining.sol#184-226) ignores return value by liquidityToken.transferFrom(msg.sender,address(this),deposit) (contracts/LiquidityMining.sol#223)
LiquidityMining.withdrawFromSlotInternal(uint256) (contracts/LiquidityMining.sol#242-249) ignores return value by liquidityToken.transfer(slot.owner,slot.deposit) (contracts/LiquidityMining.sol#246)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Although there are many Re-entrancy detections, only one detection could be a vulnerability. The rest of the detections are false positives after being checked in the code according to its functionality.

3.5 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 LiquidityMining.

Results

LiquidityMining.sol

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

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	0

THANK YOU FOR CHOOSING

HALBORN