// HALBORN

EasyFi Staking Contracts

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **June 14th - 18th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	06/10/2021	Khaled Sakr
0.2	Document Edits	06/13/2021	Khaled Sakr
1.0	Final Draft	06/13/2021	Gabi Urrutia
1.1	Remediation Plan	06/25/2021	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
Khaled Sakr	Halborn	Khaled.Sakr@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

stakingFactory is the staking implementation done by EasyFi. EasyFi engaged Halborn to conduct a security audit on their stakingFactory smart contract beginning on June 14th, 2021 and ending June 18th, 2021. The security assessment was scoped to the smart contract provided in the Github repository EasyFi Staking Smart Contract.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverable set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure smart-contract development.

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 are 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 to achieve the following:

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

In summary, Halborn identified few security risks, and 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.

1.3 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 and purpose.
- 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.
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- Testnet deployment (Truffle, Ganache)

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. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was 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

IN-SCOPE:
Code related to staking/stakingFactory.sol smart contract.

Specific commit of contract:

6d3548851c6499c2a1ea12f8a7393b0b4f34304d

Fixed commit ID:

e2b4cd89c75ce9171007bd241859aabf2adf633c

OUT-OF-SCOPE: External libraries and economics attacks

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	4

LIKELIHOOD

IMPACT

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

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PRAGMA VERSION DEPRECATED	Low	SOLVED - 06/25/2021
FLOATING PRAGMA	Low	SOLVED - 06/25/2021
MISSING BOUND CHECK	Informational	RISK ACCEPTED
INTEGER OVERFLOW	Informational	RISK ACCEPTED
NO TEST COVERAGE	Informational	SOLVED - 06/25/2021
DOCUMENTATION	Informational	SOLVED - 06/25/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) PRAGMA VERSION DEPRECATED - LOW

Description:

The current version in use for the contract is pragma ^0.5.16. While this version is still functional, and most security issues safely implemented by mitigating contracts with other utility contracts such as SafeMath.sol and ReentrancyGuard.sol, the risk to the long-term sustainability and integrity of the solidity code increases.

Code Location:

Listing 1: stakingFactory.sol (Lines 1)

```
1 pragma solidity ^0.5.16;
2
3 /**
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendations:

At the time of this audit, the current version is already at 0.8.6. When possible, use the most updated and tested pragma versions to take advantage of new features that provide checks and accounting, as well as prevent insecure use of code. (0.6.12)

Remediation Plan:

SOLVED: Pragma version was upgraded to 0.7.6.

3.2 (HAL-02) FLOATING PRAGMA - LOW

Description:

Smart contract stakingFactory.sol uses the floating pragma ^0.5.16. Locking the **pragma** helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively or recently released pragma versions may have unknown security vulnerabilities.

Code Location:

Listing 2: stakingFactory.sol (Lines 1)

```
1 pragma solidity ^0.5.16;
2
3 /**
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendations:

Consider locking the pragma version. It is not recommended to use a floating pragma in production. Apart from just locking the pragma version in the code, the sign (^) need to be removed. It is possible to lock the pragma by fixing the version both in truffle-config.js for Truffle framework or in hardhat.config.js for HardHat framework.

Remediation Plan:

SOLVED: Pragma version was locked to 0.7.6.

3.3 (HAL-03) MISSING BOUND CHECK - INFORMATIONAL

Description:

In `notifyRewardAmount()` function, `rewardRate` is calculated dividing `reward` by `RewardDuration`. In addition, `reward.add(leftover)` is divided by `rewardsDuration` to calculate `rewardRate` as well. If both denominators are greater than the numerators, `rewardRate` leads to `0`.

Code Location:

Listing 3: stakingFactory.sol (Lines 1)

```
619     function notifyRewardAmount(uint256 reward) external
            onlyRewardsDistribution updateReward(address(0)) {
620
621         if (block.timestamp >= periodFinish) {
622
623             rewardRate = reward.div(rewardsDuration);
624
625         } else {
626
627             uint256 remaining = periodFinish.sub(block.timestamp);
628
629             uint256 leftover = remaining.mul(rewardRate);
630
631             rewardRate = reward.add(leftover).div(rewardsDuration)
                    ;
632
633         }
634
635
636
637         // Ensure the provided reward amount is not more than the
                balance in the contract.
638
639         // This keeps the reward rate in the right range,
                preventing overflows due to
640
```

```
641         // very high values of rewardRate in the earned and
                rewardsPerToken functions;
642
643         // Reward + leftover must be less than 2^256 / 10^18 to
                avoid overflow.
644
645         uint balance = rewardsToken.balanceOf(address(this));
646
647         require(rewardRate <= balance.div(rewardsDuration), "
                Provided reward too high");
648
649
650
651         lastUpdateTime = block.timestamp;
652
653         periodFinish = block.timestamp.add(rewardsDuration);
654
655         emit RewardAdded(reward);
656
657     }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

Consider adding modifiers for bound checks such as:

Listing 4

```
1 require(reward >= rewardsDuration, "Reward is too small");
```

Listing 5

```
1 require(reward.add(leftover) >= rewardsDuration, "Reward is too
      small");
```

Remediation Plan:

RISK ACCEPTED: `rewardsDuration` is based on UNIX timestamp and `reward` is in wei(token with decimals 6,8,18 includes). Rewards will never be less than 1000 Tokens. With least case (Reward token with 6 decimals and duration for one year) 1,000,000,000 will be greater than 31,556,952(for 1 year).

3.4 (HAL-04) INTEGER OVERFLOW - INFORMATIONAL

Description:

An overflow happens when an arithmetic operation reaches the maximum size of a type. If the reward is higher than uint(-1).div(1e18) could happen an overflow. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits -- either larger than the maximum or lower than the minimum representable value.

Code Location:

Listing 6: stakingFactory.sol (Lines 1)

```
619     function notifyRewardAmount(uint256 reward) external
            onlyRewardsDistribution updateReward(address(0)) {
620
621         if (block.timestamp >= periodFinish) {
622
623             rewardRate = reward.div(rewardsDuration);
624
625         } else {
626
627             uint256 remaining = periodFinish.sub(block.timestamp);
628
629             uint256 leftover = remaining.mul(rewardRate);
630
631             rewardRate = reward.add(leftover).div(rewardsDuration)
                    ;
632
633         }
634
635
636
637         // Ensure the provided reward amount is not more than the
                balance in the contract.
638
```

```
639         // This keeps the reward rate in the right range,
                preventing overflows due to
640
641         // very high values of rewardRate in the earned and
                rewardsPerToken functions;
642
643         // Reward + leftover must be less than 2^256 / 10^18 to
                avoid overflow.
644
645         uint balance = rewardsToken.balanceOf(address(this));
646
647         require(rewardRate <= balance.div(rewardsDuration), "
                Provided reward too high");
648
649
650
651         lastUpdateTime = block.timestamp;
652
653         periodFinish = block.timestamp.add(rewardsDuration);
654
655         emit RewardAdded(reward);
656
657     }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

Although SafeMath library is used, consider adding the following modifier as well to prevent the overflow:

Listing 7

```
1 require(reward < uint(-1).div(1e18), "Reward overflow");
```

Remediation Plan:

RISK ACCEPTED: `rewardAmount` is set in Factory contract at deployment. Then, it can never be greater than `uint(-1).div(1e18)`.

3.5 (HAL-05) NO TEST COVERAGE - INFORMATIONAL

Description:

Unlike traditional software, smart contracts can not be modified unless deployed using a proxy contract. Because of the permanence, unit tests and functional testing are recommended to ensure the code works correctly before deployment. Mocha and Chai are valuable tools to perform unit tests in smart contracts. Mocha is a Javascript testing framework for creating synchronous and asynchronous unit tests, and Chai is a library with assertion functionality such as assert or expect and should be used to develop custom unit tests.

References:
https://github.com/mochajs/mocha
https://github.com/chaijs/chai
https://docs.openzeppelin.com/learn/writing-automated-tests

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

We recommend performing as many test cases as possible to cover all conceivable scenarios in the smart contract.

Remediation Plan:

SOLVED: EasyFi Team added test coverage.

3.6 (HAL-06) DOCUMENTATION - INFORMATIONAL

Description:

The documentation provided by the EasyFi team is not complete. For instance, the documentation included in the GitHub repository should include a walkthrough to deploy and test the smart contracts.

Recommendation:

Consider updating the documentation in Github for greater ease when contracts are deployed and tested. Have a Non-Developer or QA resource work through the process to make sure it addresses any gaps in the set-up steps due to technical assumptions.

Remediation Plan:

SOLVED: EasyFi Team documented all deployment stage.

AUTOMATED TESTING

4.1 STATIC ANALYSIS REPORT

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. 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.

Results:

```
INFO:Detectors:
StakingRewards.notifyRewardAmount(uint256) (contracts/staking/stakingFactory.sol#619-638) performs a multiplication on the result of a division:
        -rewardRate = reward.div(rewardsDuration) (contracts/staking/stakingFactory.sol#621)
        -leftover = remaining.mul(rewardRate) (contracts/staking/stakingFactory.sol#624)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

The finding can be considered a false positive, because mathematical operations are well implemented.

4.2 AUTOMATED SECURITY SCAN

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. Only security-related findings are shown below.

Results:

```
Report for contracts/staking/stakingFactory.sol
https://dashboard.mythx.io/#/console/analyses/01c2b4e8-641f-4f30-b5b5-b0fe400ba978
```

Line	SWC Title	Severity	Short Description
1	(SWC-103) Floating Pragma	Low	A floating pragma is set.
103	(SWC-000) Unknown	Medium	Function could be marked as external.
129	(SWC-000) Unknown	Medium	Function could be marked as external.
138	(SWC-000) Unknown	Medium	Function could be marked as external.
308	(SWC-000) Unknown	Medium	Function could be marked as external.
316	(SWC-000) Unknown	Medium	Function could be marked as external.
332	(SWC-000) Unknown	Medium	Function could be marked as external.
695	(SWC-000) Unknown	Medium	Function could be marked as external.
707	(SWC-000) Unknown	Medium	Function could be marked as external.
709	(SWC-128) DoS With Block Gas Limit	Medium	Loop over unbounded data structure.

THANK YOU FOR CHOOSING

// HALBORN