// HALBORN

Seascape - New Staking Saloon

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **May 11th, 2022 - May 12th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/12/2022	Alessandro Cara
0.2	Draft Review	05/13/2022	Gabi Urrutia
1.0	Remediation Plan	05/30/2022	Alessandro Cara
1.1	Remediation Plan Review	05/30/2022	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

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Seascape engaged Halborn to conduct a security audit on their smart contracts beginning on May 11th, 2022 and ending on May 12th, 2022. The security assessment was scoped to the smart contracts provided to the Halborn team.

The contract in scope was an updated version of the Staking Saloon minigame, with a new reward mechanism.

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 were mostly addressed by the Seascape 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 bridge 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 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 contract, and imported functions (Slither)
- Testnet deployment (Brownie, 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 smart contract available in GitHub at commit ID **3a438cc3d4f7dc615b0609c67785d340c820da62**.

Remediations were checked up until commit **f4d62567a58fdbcb582cb83bac36eca846917148**.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	1	4

LIKELIHOOD

IMPACT

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

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 ADMIN CAN CHANGE SESSION DETAILS AFTER THE START OF A SESSION	Medium	SOLVED - 05/30/2022
HAL-02 NOT CHECKING BALANCE BEFORE AND AFTER UNTRUSTED TOKENS TRANSFERS	Medium	RISK ACCEPTED
HAL-03 UNCHECKED CONTRACT BALANCE	Low	SOLVED - 05/30/2022
HAL-04 COMMENTED OUT CODE	Informational	SOLVED - 05/30/2022
HAL-05 POSSIBLE SIGNATURE REUSE	Informational	SOLVED - 05/30/2022
HAL-06 USING ++I CONSUMES LESS GAS THAN I++ IN LOOPS	Informational	SOLVED - 05/30/2022
HAL-07 UPGRADE TO AT LEAST PRAGMA 0.8.10	Informational	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) ADMIN CAN CHANGE SESSION DETAILS AFTER THE START OF A SESSION - MEDIUM

Description:

Testing revealed that the setType function allowed the owner of the contract to change details regarding running sessions. Should this privilege be abused or the private key of the owner be stolen, users might risk of burning their NFTs without their knowledge when unstaking. This is due to the function allowing to set the NFTs in a certain session to burn and whether users get a bonus by playing the game.

Code Location:

NewStakingSaloon.sol Lines#

Listing 1

```
1 //change type about session
2   function setType(uint256 _sessionId, bool _burn, bool _specify)
↳ external onlyOwner{
3       Session storage _session = sessions[_sessionId];
4       require(block.timestamp <= (_session.startTime + _session.
↳ period), "saloon: session end");
5       _session.burn    = _burn;
6       _session.specify = _specify;
7   }
```

Risk Level:

Likelihood - 1
Impact - 5

Recommendation:

Halborn recommends that this function is removed so that it cannot be abused by the owner role.

Remediation Plan:

SOLVED: The Seascape team removed the function.

3.2 (HAL-02) NOT CHECKING BALANCE BEFORE AND AFTER UNTRUSTED TOKENS TRANSFERS - MEDIUM

In the smart contract, if untrusted tokens are used as reward tokens, they could be manipulated in such a way that the transfer never happens. This is due to the code not checking the balance before and after the transfer.
It should be noted that reward tokens are added by the contract owner.

Code Location:

NewStakingSaloon.sol Lines# 512-540

Listing 2

```
    //Get check-in revenue
    function getSignInReward(uint256 _sessionId, uint8 _tagNum,
↳ uint8 v, bytes32 r, bytes32 s) external
        Params storage params = signinRewards[_sessionId][_tagNum
↳ ];
        Session storage _session = sessions[_sessionId];
        require(received[_sessionId][msg.sender][_tagNum] == false
↳ , "saloon: this signid reward is already received");
        require(_tagNum <= _session.signinRewardNum, "saloon: this
↳  reward do not _tagNum");
        IERC20 token = IERC20(params.token);
        uint256 NftId = 0;

        {
            bytes32 _messageNoPrefix = keccak256(abi.encodePacked(
↳ _sessionId, _tagNum, msg.sender));
            bytes32 _message = keccak256(abi.encodePacked("\
↳ x19Ethereum Signed Message:\n32", _messageNoPrefix));
            address _recover = ecrecover(_message, v, r, s);
            require(_recover == verifier,  "Verification failed");
        }

        if (params.quality > 0) {
```

```
18            NftId = nftFactory.mintQuality(msg.sender, params.
↳ generation, params.quality);
19        }
20
21        if (params.amount > 0) {
22            require(token.transferFrom(owner(), msg.sender, params
↳ .amount), "saloon: transferFrom reward failed");
23        }
24        received[_sessionId][msg.sender][_tagNum] = true;
25        emit GetSignInReward(msg.sender, _sessionId, _tagNum,
↳ NftId, params.amount);
26    }
```

Risk Level:

Likelihood - 1
Impact - 5

Recommendation:

It is recommended to always check that the balance difference before and after is the same amount requested. This will prevent untrusted tokens from manipulating balances and allow the system to ensure integrity.

Remediation Plan:

RISK ACCEPTED: The Seascape team accepted the risk, as they will only use a trusted token, Crowns.

3.3 (HAL-03) UNCHECKED CONTRACT BALANCE - LOW

Description:

Testing revealed that when a session is created, there are no checks to ensure that the contract balance is enough to pay the reward. This could lead to failed transactions when users unstake or claim their rewards.

Code Location:

NewStakingSaloon.sol Lines# 117-156

Listing 3

```
function startSession(
    address _rewardToken,
    uint256 _totalReward,
    uint256 _period,
    uint256 _startTime,
    address _verifier
)
    external
    onlyOwner
{
    require(_rewardToken != address(0), "Token can't be zero address");
    require(_startTime > block.timestamp, "Seascape Staking: Seassion should start in the future");
    require(_period > 0, "Seascape Staking: Session duration should be greater than 0");
   require(_totalReward > 0, "Seascape Staking: Reward amount should be greater than 0");
    require(_verifier != address(0), "verifier can't be zero address");

    if (lastSessionId > 0) {
        require(!isActive(lastSessionId), "Seascape Staking: Can't start when session is active");
    }
```

```
20
21      /// @dev required CWS balance of this contract
22      // require(crowns.balanceOf(address(this)) >= _totalReward, "
↳ Seascape Staking: Not enough balance of Crowns for reward");
23
24      //
↳ ------------------------------------------------------------------------
↳
25      // creating the session
26      //
↳ ------------------------------------------------------------------------
↳
27      uint256 _sessionId = sessionId.current();
28      uint256 _rewardUnit = _totalReward.mul(MULTIPLIER).div(_period
↳ );
29      sessions[_sessionId] = Session(_totalReward, _period,
↳ _startTime, 0, 0, _rewardUnit, 0, 0, _startTime, true, false, 0);
30
31      //
↳ ------------------------------------------------------------------------
↳
32      // updating rest of session related data
33      //
↳ ------------------------------------------------------------------------
↳
34      sessionId.increment();
35      rewardToken = IERC20(_rewardToken);
36      lastSessionId = _sessionId;
37      verifier = _verifier;
38
39      emit SessionStarted(_sessionId, _totalReward, _startTime,
↳ _startTime + _period);
40  }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Halborn recommends that a check is added to ensure that the contract balance is enough to pay rewards to the users. This could be implemented as follows:

Listing 4

```
require(rewardToken.balanceOf(address(this)) >= _totalReward,  "
↳ Seascape Staking: Not enough balance for reward");
```

Remediation Plan:

SOLVED: The Seascape team amended the code to add contract balance checks.

3.4 (HAL-04) COMMENTED OUT CODE - INFORMATIONAL

Description:

There are instances within the code where commented code is left from previous development iterations. While this does not cause any security concerns, it makes the contract less readable.

Code Location:

NewStakingSaloon.sol Lines# 137-138

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Halborn recommends that the code is removed to improve the readability of the code.

Remediation Plan:

SOLVED: The Seascape team removed commented code for readability.

3.5 (HAL-05) POSSIBLE SIGNATURE REPLAY - INFORMATIONAL

Description:

Within the NewStakingSaloon.sol smart contract, there are some occurrences where signature replay attacks can be executed. While the contract logic would prevent any meaningful exploitation of those, there might be some extreme test cases which might introduce risks to the platform and their users.

Code Location:

NewStakingSaloon.sol Lines# 196-201

- In the deposit function

Listing 5

```
1   {
2     bytes32 _messageNoPrefix = keccak256(abi.encodePacked(_nftId,
↳ _sp));
3     bytes32 _message = keccak256(abi.encodePacked("\x19Ethereum
↳ Signed Message:\n32", _messageNoPrefix));
4     address _recover = ecrecover(_message, _v, _r, _s);
5     require(_recover == verifier,  "Nft Staking: Seascape points
↳ verification failed");
6 }
```

- In the verifyBonus function

Listing 6

```
1 bytes32 _messageNoPrefix = keccak256(abi.encodePacked(
2     _bonusPercent,
3     _balance[0].nftId,
```

```
    _balance[1].nftId,
    _balance[2].nftId
));

/// Validation of bonus
/// @dev 3. verify that signature for message was signed by contract owner
  bytes32 _message = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", _messageNoPrefix));
  address _recover = ecrecover(_message, _v, _r, _s);
  require(_recover == verifier, "NFT Staking: Seascape points verification failed");
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to have a unique value in the signatures that always prevents the reuse of signatures. In this case, it is recommended to use a nonce value that would be stored on an account basis or to use `msg.sender` in the signature verification. This will prevent any other user from reusing previous signatures.

Remediation Plan:

SOLVED: The Seascape team added a nonce to the signature verification.

3.6 (HAL-06) USING ++I CONSUMES LESS GAS THAN I++ IN LOOPS - INFORMATIONAL

Description:

In the loop below, the variable `i` is incremented using `i++`. It is known that, in loops, using `++i` costs less gas per iteration than `i++`. This does not only apply to the iterator variable. It also applies to variables declared within the loop code block.

Code Location:

NewStakingSaloon.sol Line #372-374

Listing 7

```
1    for(uint _index = 0; _index < 3; _index++){
2         _interests = _interests.add(calculateInterest(_sessionId,
↳ msg.sender, _index));
3     }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to use `++i` instead of `i++` to increment the value of an `uint` variable inside a loop. This applies to the iterator variable and to variables declared within the loop code block.

Remediation Plan:

SOLVED: The Seascape team amended the code as suggested to optimize the contract.

3.7 (HAL-07) UPGRADE TO AT LEAST PRAGMA 0.8.10 - INFORMATIONAL

Description:

Gas optimizations and additional safety checks are available for free when using newer compiler versions and the optimizer.

- Safemath by default since 0.8.0 (can be more gas efficient than the SafeMath library)
- Low level inline: as of 0.8.2, leads to cheaper gas runtime. This is especially relevant when the contract has small functions. For example, OpenZeppelin libraries typically have a lot of small helper functions and if they are not built in, they cost an additional 20 to 40 gas due to the 2 extra jump instructions and additional stack operations needed for function calls.
- Optimizer improvements in packed structs: Before 0.8.3, storing packed structs, in some cases, used an additional storage read operation. After EIP-2929, if the slot was already cold, this means unnecessary stack operations and extra deploy time costs. However, if the slot was already warm, this means additional cost of 100 gas alongside the same unnecessary stack operations and extra deploy time costs.
- Custom errors from 0.8.4, leads to cheaper deploy time cost and run time cost. Note: the run time cost is only relevant when the revert condition is met. In short, replace revert strings by custom errors.

Code Location:

The contract within scope made use of the pragma version 0.6.7

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Halborn recommends that the project is upgraded to use at least **pragma** 0.8.10.

Remediation Plan:

ACKNOWLEDGED: The Seascape team keeps the same version of the Solidity compiler.

THANK YOU FOR CHOOSING

// HALBORN