CERTIK

Preliminary Comments

Seascape Game 3

Apr 20th, 2021

Summary

This report has been prepared for Seascape Game 3 smart contracts, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases given they are currently missing in the repository;
- Provide more comments per each function for readability, especially contracts are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	Seascape Game 3
Platform	Ethereum
Language	Solidity
Codebase	https://github.com/blocklords/seascape-smartcontracts/tree/main/contracts/game_3
Commits	a5c2c2bb1f083ec417608250ddee79f2d61e9b75

Audit Summary

Delivery Date	Apr 20, 2021
Audit Methodology	Static Analysis, Manual Review
Key Components	

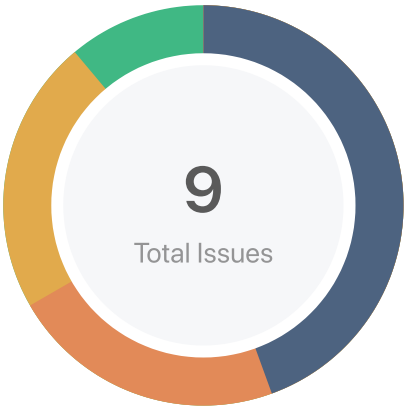
Vulnerability Summary

Total Issues	9
Critical	0
Major	2
Minor	2
Informational	4
Discussion	1

Audit Scope

ID	file	SHA256 Checksum
NSS	NftStaking.sol	54890d171839ed27a4acb45cdd07898d3073d799270a3cfc4def284f6e8f67cc

Findings



Critical	0 (0.00%)
Major	2 (22.22%)
Minor	2 (22.22%)
Informational	4 (44.44%)
Discussion	1 (11.11%)

ID	Title	Category	Severity	Status
NSS-01	Misleading Variable Name	Optimization	Informational	Pending
NSS-02	Divide before multiply	Volatile Code	Informational	Pending
NSS-03	The Receiver of Function `payDebt`	Logical Issue	Major	Pending
NSS-04	Redundant Codes	Optimization	Informational	Pending
NSS-05	Inconsistent Logic	Logical Issue	Minor	Pending
NSS-06	Lacking Time Validation	Logical Issue	Minor	Pending
NSS-07	Compares to a Boolean Constant	Optimization	Informational	Pending
NSS-08	Calculation of Session's `claimed`	Logical Issue	Major	Pending
NSS-09	Multiple Calls to Function `updateInterestPerPoint`	Optimization	Discussion	Pending

NSS-01 | Misleading Variable Name

Category	Severity	Location	Status
Optimization	Informational	NftStaking.sol: 50, 194	Pending

Description

Variable `claimedPerPoint` of struct `Balance` has different meaning with variable `claimedPerPoint` of struct `Session` as below

```
_balances[_index].claimedPerPoint =
_session.claimedPerPoint.mul(_balances[_index].sp).div(scaler);
```

Recommendation

Consider renaming `claimedPerPoint` of struct `Balance` to `claimedAmount` or some better name.

NSS-02 | Divide before multiply

Category	Severity	Location	Status
Volatile Code	Informational	NftStaking.sol: 129, 425, 345	Pending

Description

Divide as below:

```
129  uint256 _rewardUnit = _totalReward.div(_period);
```

then, multiply as below:

```
425  _session.interestPerPoint = _session.rewardUnit.mul(scaler).div(_session.totalSp);
```

Another example:

```
345  uint256 totalBonus = _interests.mul(scaler).div(100).mul(_bonusPercent).div(scaler);
```

Recommendation

Consider multiply before divide as below:

```
129  uint256 _rewardUnit = _totalReward.mul(scaler).div(_period);
```

```
425  _session.interestPerPoint = _session.rewardUnit.div(_session.totalSp);
```

and,

```
345  uint256 totalBonus = _interests.mul(scaler).mul(_bonusPercent).div(100).div(scaler);
```

NSS-03 | The Receiver of Function `payDebt`

Category	Severity	Location	Status
Logical Issue	Major	NftStaking.sol: 156	Pending

Description

Why the receiver of function `payDebt` is `address(this)` instead of passed in parameter user's `address`, is it designed as exptected?

NSS-04 | Redundant Codes

Category	Severity	Location	Status
Optimization	Informational	NftStaking.sol: 213~217	Pending

Description

Redundant codes as below:

```
if(earning[_sessionId][msg.sender] > 0){
    earning[_sessionId][msg.sender] = earning[_sessionId][msg.sender].add(_claimed);
} else{
    earning[_sessionId][msg.sender] = _claimed;
}
```

since the default value `earning[_sessionId][msg.sender]` is an unit type, its value is zero in the `else` block.

Recommendation

Consider refactoring codes as below:

```
earning[_sessionId][msg.sender] = earning[_sessionId][msg.sender].add(_claimed);
```

NSS-05 | Inconsistent Logic

Category	Severity	Location	Status
Logical Issue	Minor	NftStaking.sol: 213~217	Pending

Description

The variable `earning` stores earning of user in the `claim` function but no in the `claimAll` function, is that designed as expected?

NSS-06 | Lacking Time Validation

Category	Severity	Location	Status
Logical Issue	Minor	NftStaking.sol: 396~398	Pending

Description

Lack time validation on the case that the session not yet starts.

Recommendation

Consider adding time validation as below:

```
if (now < sessions[_sessionId].startTime || now > sessions[_sessionId].startTime +
sessions[_sessionId].period) {
    return false;
}
```

In the other side, a session is initialized as below:

```
130  sessions[_sessionId] = Session(_totalReward, _period, _startTime, 0, 0, _rewardUnit, 0, 0,
     _startTime);
```

It implies that session's `lastInterestUpdate` is assigned to be `_startTime`. When someone deposit before start time, the _sessionCap is current time that is less than start time, and session.lastInterestUpdate is start time , it will result in reverting in the calculation of `claimedPerPoint` in function `updateInterestPerPoint`:

```
417  _session.claimedPerPoint = _session.claimedPerPoint.add(
418      _sessionCap.sub(_session.lastInterestUpdate).mul(_session.interestPerPoint));
```

NSS-07 | Compares to a Boolean Constant

Category	Severity	Location	Status
Optimization	Informational	NftStaking.sol: 396~398	Pending

Description

Compares to a boolean constant.

```
396 if (isActive(_sessionId) == false) {
397     _sessionCap = _session.startTime.add(_session.period);
398 }
```

Recommendation

Consider removing the equality to the boolean constant.

NSS-08 | Calculation of Session's `claimed`

Category	Severity	Location	Status
Logical Issue	Major	NftStaking.sol: 352~368	Pending

Description

Below is the calculation of a session's `claimed`:

```
352 function transfer(uint256 _sessionId, uint256 _index) internal returns(uint256) {
353     Session storage _session = sessions[_sessionId];
354
355     uint256 _interest = calculateInterest(_sessionId, msg.sender, _index);
356
357     uint256 _crownsBalance = crowns.balanceOf(address(this));
358     if (_interest > 0 && _interest > _crownsBalance) {
359         debts[msg.sender] = _interest.sub(_crownsBalance).add(debts[msg.sender]);
360         crowns.transfer(msg.sender, _crownsBalance);
361     } else {
362         crowns.transfer(msg.sender, _interest);
363     }
364
365     _session.claimed = _session.claimed.add(_interest);
366
367     return _interest;
368 }
```

However, it is supposed to be: function transfer(uint256 _sessionId, uint256 _index) internal returns(uint256) { Session storage _session = sessions[_sessionId];

```
uint256 _interest = calculateInterest(_sessionId, msg.sender, _index);

uint256 _crownsBalance = crowns.balanceOf(address(this));
if (_interest > 0 && _interest > _crownsBalance) {
    debts[msg.sender] = _interest.sub(_crownsBalance).add(debts[msg.sender]);
    crowns.transfer(msg.sender, _crownsBalance);
    _session.claimed = _session.claimed.add(_crownsBalance);
} else {
    crowns.transfer(msg.sender, _interest);
    _session.claimed = _session.claimed.add(_interest);
}

return _interest;
```

}

NSS-09 | Multiple Calls to Function `updateInterestPerPoint`

Category	Severity	Location	Status
Optimization	Discussion	NftStaking.sol: 352~368, 162~197, 201~227	Pending

Description

Many calls to the function `updateInterestPerPoint` in functions `deposit`, `claim` and `claimAll`.

Recommendation

Consider calling the function `updateInterestPerPoint` as the first statement of the calling function after input data validation instead of calling it multiple times in a single function.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a struct assignment operation affecting an in-memory struct rather than an in storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete .

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as constant contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

About

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.