TECHFUND Audit Score

Smart contract Audit Report

CRYPTO GAMES IDOL Audit

Completed on 2022-07-22

Score **POSITIVE**

Risk level		
	Critical	0
	High	0
	Medium	0
	Low	1
	Note	2

Risk level detail

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

The tester arrives at the likelihood and impact estimates, they can now combine them to get a final severity rating for this risk. Note that if they have good business impact information, they should use that instead of the technical impact information.

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Vulnerability Review

	Number of warnings
Compiler Version	1
State Variable Default Visibility	0
Function Visibility / Gas Optimisation	1
In-Correct Mathematical Operation sequence	1
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Callstack Depth Attack	0
Production Node modules security	0
Development Node modules security	0
Re-Entrancy	0
Double Withdrawal	0

TECHFUND Audit Score

Smart contract Audit Report

Compiler Version

1

Solidity version used is inconsistent & not fixed. It is highly recommended to update the Solidity version to the latest. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements and using a fixed version of Solidity upon understanding the version specifications.

```
pragma solidity ^0.8.6;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
```

Function Visibility / Gas Optimisation

1

Functions like getNextVestingId and getAvailableAmountAggregated should be declared as external instead of public. In case of public functions, Solidity immediately copies array arguments to memory, while external functions can read directly from calldata. Memory allocation is expensive, whereas reading from calldata is cheap.
The reason that public functions need to write all of the arguments to memory is that public functions may be called internally, which is actually an entirely different process than external calls.

```
function getNextVestingId(address _beneficiary)
  public
  view
  returns (uint256)
{
  return vestingMap[_beneficiary].length;
}
```

In-Correct Mathematical Operation

1

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision.
In the case of JavaScript, following issue happens due to floating point errors. In Solidity we don't have floating points, but we get a similar issue and we can get "rounding errors".
For example : In Solidity the second operation would actually yield 2999. By doing all our multiplications first, we mitigate rounding related issues as much as possible. Hence it is suggested to do all multiplication before division otherwise the output can be in-correct.

```
» console.log((30 * 100 * 13) / 13)
  3000
← undefined
» console.log((30 / 13) * 100 * 13)
  2999.9999999999995
← undefined
»
```

```
function getAvailableAmountAtTimestamp(
  address _beneficiary,
  uint256 _vestingId,
  uint256 _timestamp
) public view returns (uint256) {
  if (_vestingId >= vestingMap[_beneficiary].length) {
    return 0;
  }

  Vesting memory vesting = vestingMap[_beneficiary][_vestingId];

  uint256 periodsPassed = _timestamp.sub(vesting.startedAt).div(90 days); // We say that 3 months are always 90 days

  uint256 alreadyReleased = vesting.releasedAmount;

  if (periodsPassed >= STEPS_AMOUNT) {
    return vesting.totalAmount.sub(alreadyReleased);
  }

  uint256 rewardPerPhase1Period = vesting.totalAmount.mul(7).div(100);   1
  uint256 phase1Peroids = 12;
  if (periodsPassed <= phase1Peroids) {
    return rewardPerPhase1Period.mul(periodsPassed).sub(alreadyReleased);
  } else {
    uint256 rewardPerPhase2Period = vesting.totalAmount.mul(2).div(100);
    uint256 phase2PeriodsPassed = periodsPassed.sub(phase1Peroids);
    uint256 rewardPhase1 = rewardPerPhase1Period.mul(phase1Peroids);   2
    uint256 rewardPhase2 = rewardPerPhase2Period.mul(phase2PeriodsPassed);
    return rewardPhase1.add(rewardPhase2).sub(alreadyReleased);
  }
}
```

```
function getAvailableAmountAtTimestamp(
  address _beneficiary,
  uint256 _vestingId,
  uint256 _timestamp
) public view returns (uint256) {
  if (_vestingId >= vestingMap[_beneficiary].length) {
    return 0;
  }

  Vesting memory vesting = vestingMap[_beneficiary][_vestingId];

  uint256 periodsPassed = _timestamp.sub(vesting.startedAt).div(90 days); // We say that 3 months are always 90 days

  uint256 alreadyReleased = vesting.releasedAmount;

  if (periodsPassed >= STEPS_AMOUNT) {
    return vesting.totalAmount.sub(alreadyReleased);
  }

  uint256 rewardPerPhase1Period = vesting.totalAmount.mul(7).div(100);
  uint256 phase1Peroids = 12;
  if (periodsPassed <= phase1Peroids) {
    return rewardPerPhase1Period.mul(periodsPassed).sub(alreadyReleased);
  } else {
    uint256 rewardPerPhase2Period = vesting.totalAmount.mul(2).div(100);   1
    uint256 phase2PeriodsPassed = periodsPassed.sub(phase1Peroids);
    uint256 rewardPhase1 = rewardPerPhase1Period.mul(phase1Peroids);
    uint256 rewardPhase2 = rewardPerPhase2Period.mul(phase2PeriodsPassed);   2
    return rewardPhase1.add(rewardPhase2).sub(alreadyReleased);
  }
}
```