SPORTS ICON

Vesting

Smart Contract Audit Report

November 24, 2021

Introduction

1. About SportIcon

SportsIcon creates cinematic NFTs in collaboration with athletes that commemorate the journey through their lives and sporting careers that made them the icons they are. While regular digital content can be shared unlimitedly online, digital content in the form of an NFT becomes a unique asset that has provenance, originality, and authenticity.

In addition to NFTs, SportsIcon is working to build the sports metaverse — a niche of the broader metaverse dedicated to sports fans and players. Metaverses are virtual worlds where individuals can connect with others. We often think about Virtual Reality metaverses, but this doesn't necessarily have to be the case — any online game or social network has a piece of a metaverse in it, although the true potential of these digital connections is only now being fully explored.

Visit https://sportsicon.com/ to know more about.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The SportIcon team has provided the following doc for the purpose of audit:

1. https://docs.sportsicon.com/

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: SportIcon
- Contracts Name: SportsIconPrivateVesting.sol
- Languages: Solidity(Smart contract)
- Github commit/Smart Contract Address for audit: 8a26ed5c3a52b7a4d1b22a38b40dd80783ecc6a3
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	2
Closed	-	-	-

High Severity Issues

No issues were found.

Medium Severity Issues

No issues were found.

Low Severity Issues

1. **Return Value of an External Call is never used Effectively**
   Line no - 60

   Explanation:
   The external calls made in the above-mentioned lines do return a boolean value that indicates whether or not the external call made was successful.

```
56      function claim() external override returns (uint256) {
57          uint256 tokens = freeTokens(msg.sender);
58          claimedOf[msg.sender] = claimedOf[msg.sender].add(tokens);
59
60          token.transfer(msg.sender, tokens);
61
62          emit LogTokensClaimed(msg.sender, tokens);
63
64          return tokens;
65      }
```

   These boolean return values can be used in the function as a check to ensure that the further execution of the function is only allowed if the external is successfully made.
   However, the **SportsIconPrivateVesting** contract never uses these return values throughout the contract.

   Recommendation:
   Effective use of all the return values from external calls must be ensured within the contract.

2. **Unused local return variable found in contract function**
   Line no - 67

   Explanation:
   The **calculateOwed()** function in the contract includes a return variable, i.e., **owed**.

```
66
67      function calculateOwed(address user) internal view returns (uint256 owed) {
68          if (vestedTokensOfPrivileged[user] > 0) {
69              return vestedTokensOfPrivileged[user];
70          }
```

   However, it was found during the audit that the variable was not used in the return statements of the function.

   Recommendation:
   If a specific local or state variable is not supposed to be used in the function or contract, it should be removed to improve the efficiency as well as readability.

Recommendations

1. **Contract might completely lock any additional tokens transferred to it**
   Explanation:

   As per the current architecture of the vesting contract, the only function to withdraw tokens out of the contract is the **claim()** function.

   It must be noted that the **claim()** function calculates the exact amount that should be transferred to the user calling the function.

   However, if any specific amount of additional token is sent to the contract, it might get locked completely as there is no other way of withdrawing tokens from the contract except the claim function.

   Recommendation:
   The above-mentioned issue doesn't break the intended behavior of the contract as of now. However, it should be kept in mind and appropriate functions should be added in case there are chances of such mistakes.

   Additionally, the tokens assigned to each address during the contract deployment should be carefully verified as the contract doesn't provide any mechanism to update assigned tokens later.

2. **Test Cases can be improved.**
   Explanation:
   The current test cases provided for the vesting contract include a few imperative scenarios which test the behavior of the code.

   However, the test cases do not cover all the possible scenarios of the contract. Most of the vesting amount calculation and the claiming procedure are tested for one specific user, i.e., **Alice** with an exact token value of **700** tokens.

   Recommendation:
   Although the current test cases do check some crucial features of the contract, it shall be quite effective to include test cases that check for different token values as well as cover the edge cases that might appear during the contract execution.

Automated Audit Result

1. SportsIconPrivateVesting.col

```
Compiled with solc
Number of lines: 387 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 4 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 0
Number of informational issues: 9
Number of low issues: 2
Number of medium issues: 1
Number of high issues: 1
ERCs: ERC20

+-------------------------+-------------+-------+-------------------+--------------+--------------------+
|          Name           | # functions | ERCS  |     ERC20 info    | Complex code |      Features      |
+-------------------------+-------------+-------+-------------------+--------------+--------------------+
|         IERC20          |      6      | ERC20 |     No Minting    |      No      |                    |
|                         |             |       | Approve Race Cond.|              |                    |
|                         |             |       |                   |              |                    |
|        SafeMath         |      13     |       |                   |      No      |                    |
| SportsIconPrivateVesting |      9      |       |                   |      No      | Tokens interaction |
+-------------------------+-------------+-------+-------------------+--------------+--------------------+
INFO:Slither:flat.sol analyzed (4 contracts)
```

Test Cases

```
SportsIconPrivateVesting contract
  ✓ Should deploy Vesting (50ms)
  Vesting Calculations
    ✓ Should calculate free tokens
    ✓ Should claim freely initial tokens
    ✓ Should claim freely initial tokens as privileged investor
    ✓ Should claim freely initial tokens + monthly together
    ✓ Should claim freely initial tokens and monthly separately
    ✓ Should claim all tokens in the end
    ✓ Should have no tokens after all claim


8 passing (744ms)
```

Concluding Remarks

While conducting the audits of the SportIcon smart contract, it was observed that the contracts contain Low severity issues.

Our auditors suggest that Low severity issues should be resolved by the developers. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the SportIcon platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes