QuillAudits

Audit Report
January, 2022

For

jUmptask

Contents

Contents

Overview

JumpTask is a gig economy-based marketplace that allows companies and organizations to make the most out of the collective skills possessed by a globally dispersed workforce. In addition, it allows gig workers to discover and explore the cryptoworld, as all the payments on the platform are made in its own cryptocurrency – JumpToken (JMPT).

Scope of the Audit

The scope of this audit was to analyse JumpTask smart contract's codebase for quality, security, and correctness.

JumpTask Contracts:
JumpToken
JumpTokenTimelock

Final contracts reflecting the bug fixes reported during the audit:
JumpToken
JumpTokenTimelock

Note: There will be multiple copies of JumpTokenTimelock with different release times to support the use case and business logic of the project. One such reference is attached above.

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries

- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20/BEP20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	0	2
Closed	0	1	3	2

Functional Testing Results

Some of the tests performed are mentioned below

JumpToken

- User should be able to burn its tokens. PASS
- User should be able to transfer its tokens. PASS
- Spender should be able to burn the owner's tokens, considering the token allowance given by the owner. PASS
- Spender should be able to transfer the owner's tokens, considering the token allowance given by the owner. PASS
- Owner should be able to change the allowance given to the spender any time. PASS

JumpTokenTimelock

- Should deploy with correct values. PASS
- Should send the locked token liquidity to the beneficiary address once the releaseTime has passed. PASS

Issues Found

High severity issues

No issues were found.

Medium severity issues

- **Missing Important Checks**

JumpTokenTimelock implements openzeppelin's TokenTimelock in order to implement a simple vesting mechanism.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

import "@openzeppelin/contracts/token/ERC20/utils/TokenTimelock.sol";

contract JumpTokenTimelock is TokenTimelock {
    constructor(IERC20 token, address beneficiary, uint256 releaseTime)
    public
    TokenTimelock(token, beneficiary, releaseTime)
    {}
}
```

However, it lacks important checks, which may lead to incorrect contract initialization.

The missing checks are:

1. Zero address check for token and beneficiary addresses: Zero address initialization of token will make the contract unoperational and meaningless.
2. The contract lacks value checks for releaseTime, as a result the deployer may accidentally pass a value, much sooner than intended or far away in future(for instance 2**256-1), thus locking the token liquidity forever.

Recommendation
Consider adding required checks.

Status: Fixed

Low severity issues

- **Do Not comply with BEP20 standard completely**

  BEP20 standard makes it mandatory for the tokens to define a function as getOwner, which should return the owner of the token contract.

5.1.1.6 getOwner

```
function getOwner() external view returns (address);
```

- Returns the bep20 token owner which is necessary for binding with bep2 token.
- NOTE - This is an extended method of EIP20. Tokens which don't implement this method will never flow across the Binance Chain and Binance Smart Chain.

However, the token contract doesn't implement/define any such function, as a result the token may not flow across the Binance Chain and Binance Smart Chain, as stated by BEP20 interface documentation.

Recommendation
Consider adding the getOwner function.

Status: Fixed

- **Unnecessary/Unused SafeERC20 library**

  JumpToken implements SafeERC20 library. SafeERC20 by openzeppelin provides wrappers around ERC20 operations, to support multiple variants of ERC20 tokens in existence.

```
2  pragma solidity ^0.8.2;
3
4  import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
5  import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
6  import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
7
8  contract JumpToken is ERC20, ERC20Burnable {
9    using SafeERC20 for IERC20;
10
```

However, JumpToken does not make use of any such feature provided by SafeERC20

Recommendation
Consider reviewing and verifying the business and operational logic, and removing the SafeERC20 implementation as the library, in order to reduce the bytecode and deployment cost, and increase the code readability. SafeERC20 helps to safely interact with a third party token. It is not meant to be used inside the token contract itself.

Status: Fixed

- **SafeERC20 vs SafeBEP20**

  JumpTokenTimelock implements openzeppelin's TokenTimelock in order to implement a simple vesting mechanism. TokenTimelock implements SafeERC20 library and IERC20 interface. However, JumpTokenTimelock is intended to work with BEP20 tokens. Although the operational logic of both OZ's SafeERC20 and Pancake-Swap's SafeBEP20 is the same, still, not using the right library and interface reduces the code readability.

  Recommendation
  Consider reviewing and verifying the business and operational logic, and using SafeBEP20 for BEP20 operations in order to increase code readability.

  Status: Fixed

Informational issues

- **Unnecessary Constructor visibility**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

import "@openzeppelin/contracts/token/ERC20/utils/TokenTimelock.sol";

contract JumpTokenTimelock is TokenTimelock {
    constructor(IERC20 token, address beneficiary, uint256 releaseTime)
    public
    TokenTimelock(token, beneficiary, releaseTime)
    {}
}
```

JumpTokenTimelock defines a public visibility for the constructor. However, from 0.7 onwards, visibility for constructors is not needed anymore, and hence can be removed.

Reference
https://docs.soliditylang.org/en/v0.7.0/070-breaking-changes.html#functions-and-events

Recommendation
Consider removing the public keyword as the visibility concept for constructors is now obsolete from 0.7 onwards

Status: Fixed

- **Centralization Risk**

  JumpToken is a fixed supply token and mints the entire 100M token liquidity to the deployer of the contract.

```
contract JumpToken is ERC20, ERC20Burnable {
  using SafeERC20 for IERC20;

  constructor() ERC20("JumpToken", "JMPT") {
    // Mint 100M tokens to creator address
    _mint(msg.sender, 100000000 * 10 ** decimals());
  }
}
```

  The implementation is prone to centralization risk that may arise, if the deployer loses its private key.

  Status: Acknowledged
  DEV Comments: “The total supply will be transferred from initial/ deployer’s address to JumpTokenTimelock”

- **Multiple pragma directives have been used.**

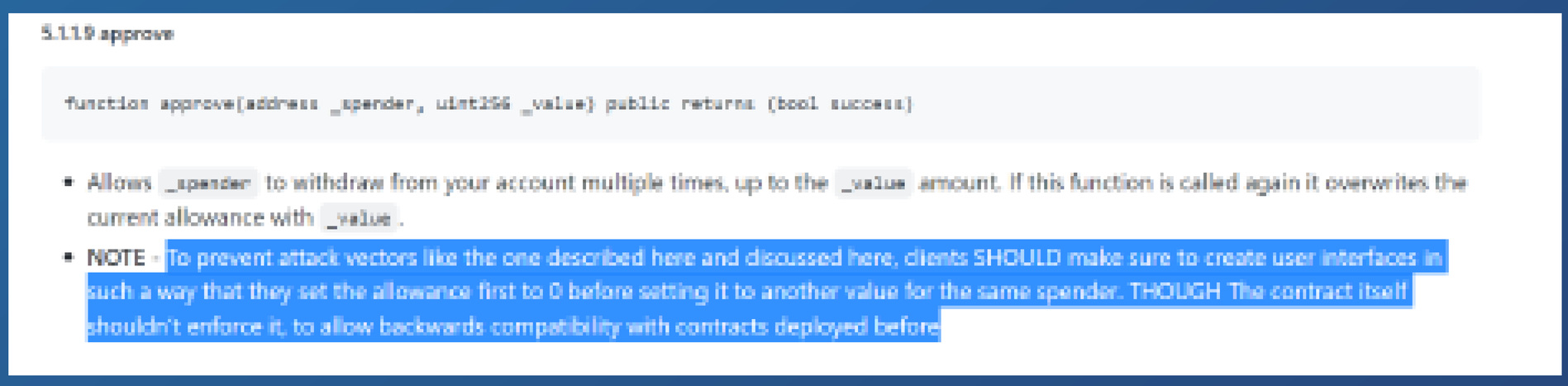
  Recommendation
  Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in pragma solidity 0.5.10) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. Ref: Security Pitfall 2

  Status: Fixed

- **BEP20 approve race condition**

  The standard ERC20 implementation contains a widely-known racing condition in its approve function, wherein a spender is able to witness the token owner broadcast a transaction altering their approval and quickly sign and broadcast a transaction using transferFrom to move the current approved amount from the owner's balance to the spender. If the spender's transaction is validated before the owner's, the spender is able to spend their entire approval amount twice.

  As the BEP20 standard is proposed by deriving the ERC20 protocol of Ethereum, the race condition exists here as well.



5.1.19 approve

```
function approve(address _spender, uint256 _value) public returns (bool success)
```

- Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.
- NOTE - To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

Reference

1. https://eips.ethereum.org/EIPS/eip-20
2. https://github.com/binance-chain/BEPs/blob/master/BEP20.md#5119-approve

Status: Acknowledged

Closing Summary

Some issues of Medium and Low severity were found. All of them have now been fixed by the project team. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the JumpTask contract. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the JumpTask team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
January, 2022

For

jUmptask

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com