



SMART CONTRACT AUDIT REPORT

for

Jungle Road Token



Prepared By: Patrick Liu

PeckShield
March 1, 2022

Document Properties

Client	Jungle Road
Title	Smart Contract Audit Report
Target	Jungle Road Token
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	March 1, 2022	Luck Hu	Final Release
1.0-rc	February 22, 2022	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Liu
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Jungle Road Token	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	8
2.1	Summary	8
2.2	Key Findings	9
3	ERC20 Compliance Checks	10
4	Detailed Results	13
4.1	Suggested immutable in multiSigWallet For Gas Efficiency	13
4.2	Trust Issue Of Admin Keys	14
5	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related source code of the `Jungle Road Token` contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to either security or performance. This document outlines our audit results.

1.1 About Jungle Road Token

`Jungle Road Token` is an ERC20-compliant protocol token on `Binance Smart Chain (BSC)` blockchain. It is a governance token for `Jungle Road` which can be used to vote for mechanism change, purchase new NFTs, refill `Stamina` and heal the character, play lottery, and participate in a variety of community events. The basic information of the audited token contract is as follows:

Table 1.1: Basic Information Of Jungle Road Token

Item	Description
Issuer	Jungle Road
Website	https://www.jungleroad.io/
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	March 1, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- <https://github.com/Jungle-Road/jungleroad-contract.git> (b58252b)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/Jungle-Road/jungleroad-contract.git> (22cd541)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead of Transfer
	Costly Loop
	(Unsafe) Use of Untrusted Libraries
	(Unsafe) Use of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe

regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table [1.3](#).

1.4 Disclaimer



Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Jungle Road Token` contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	0	
Informational	1	
Total	2	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found, and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Jungle Road Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Suggested immutable in multiSigWallet For Gas Efficiency	Coding Practices	Fixed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 4 for details.



3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `view-only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited Jungle Road Token. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	—
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to <i>address(0x0)</i> when new tokens are generated	—
Approval() event	Is emitted on any successful call to approve()	✓

adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on <code>transfer()/transferFrom()</code> calls	—
Rebasing	The <code>balanceOf()</code> function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	—

4 | Detailed Results

4.1 Suggested immutable in multiSigWallet For Gas Efficiency

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: JungleRoadToken
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, [Solidity](#) introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., “a constant, once assigned in the constructor, is read-only during the subsequent operation.”

In the following, we show one key state variable `multiSigWallet` defined in `JungleRoadToken` contract. If there is no need to dynamically update it after the construction, it can be declared as either constants or `immutable` for gas efficiency. In particular, the state variable `multiSigWallet` can be defined as `immutable` as it will not be changed after its initialization in `constructor()`.

```
8 contract JungleRoadToken is ERC20, Pausable {
9     address public multiSigWallet; // create by gnosis safe
10
11     /**
12      * @notice Require that the sender is the multiSigWallet.
13      */
```

```

14     modifier onlyMultiSigWallet() {
15         require(msg.sender == multiSigWallet, "Why do you do that");
16         _;
17     }
18
19     constructor(address _multiSigWallet) public ERC20("Jungle Road Token", "JGRD") {
20         // 1000M total supply of tokens
21         require(_multiSigWallet != address(0), "multiSigWallet cannot be the zero address");
22         multiSigWallet = _multiSigWallet;
23
24         _mint(_multiSigWallet, 1000000000 ether); // 1,000,000,000 ether
25     }
26     ...
27 }

```

Listing 4.1: JungleRoadToken.sol

Recommendation Revisit the state variable definition and make extensive use of `constant/immutable` states.

Status The issue has been fixed by this commit: 22cd541.

4.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: JungleRoadToken
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In Jungle Road Token contract, there is a privileged `multiSigWallet` account (assigned in the `constructor()`) that plays a critical role in governing and regulating the token-related operations.

```

27     function pause() public onlyMultiSigWallet {
28         _pause();
29     }
30
31     function unpause() public onlyMultiSigWallet {
32         _unpause();
33     }

```

Listing 4.2: JungleRoadToken::pause()/unpause()

To elaborate, we show above the sensitive operations that are related to `multiSigWallet`. Specifically, it has the authority to pause/unpause JGRD token transfer. It would be worrisome if the `multiSigWallet` account is a plain EOA account. The current multi-sig account greatly alleviates this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

Recommendation Promptly transfer the `multiSigWallet` privilege of JGRD token to the intended governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by customer.



5 | Conclusion

In this security audit, we have examined the design and implementation of the `Jungle Road Token` contract. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified two issues of varying severities. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.



References

- [1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. <https://cwe.mitre.org/data/definitions/1099.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [6] PeckShield. PeckShield Inc. <https://www.peckshield.com>.