SMART CONTRACT AUDIT REPORT

for

RowaToken & RowaVesting

Prepared By: Xiaomi Huang

PeckShield
April 12, 2023

Document Properties

Client	ROWA
Title	Smart Contract Audit Report
Target	ROWA
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	April 12, 2023	Xuxian Jiang	Final Release
1.0-rc	March 31, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the ROWA 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 BEP20-compliance, security, or performance. This document outlines our audit results.

1.1 About ROWA

ROWA platform is a GameFi project designed to create a secure and sustainable ecosystem for gamers and game creators. ROWA platform has a token named $ROWA with a maximum supply of one billion, which will be launched on Polygon. The audit evalutes the ERC20-compliance of $ROWA as well as the security of associated vesting contract RowaVesting. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of ROWA

Item	Description
Issuer	ROWA
Website	https://rowa.games/
Type	ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 12, 2023

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

- https://github.com/rowagames/ROWA-Token.git (cf5a9b9)

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

- https://github.com/rowagames/ROWA-Token.git (2ecddd5)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

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.
- BEP20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard BEP20 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
BEP20 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 ROWA 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	2	■ ■
Informational	0	
Total	3	

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. Overall, there is no critical or high severity issue, although the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key ROWA Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Cliff Consideration in _computeReleasableAmount()	Business Logic	Resolved
PVE-002	Low	Improved Revoke Logic in RowaVesting	Coding Practices	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Resolved

Besides recommending specific countermeasures to mitigate the above issue(s), we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

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 `ROWA` token contract. 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-adopted ERC20 specification.

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 $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

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, 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 transfer()/transferFrom() calls	—
Rebasing	The balanceOf() 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 Revisited Cliff Consideration in _computeReleasableAmount()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RowaVesting
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The RowaVesting contract aims to instantiate different vesting schedules. By design, the vesting schedules ensure that assets are locked and released slowly over a certain period of time. In other words, it means the assets can't be sold, transferred, or transacted until they are released. Moreover, there is a notion of cliff, which is the period of time that must pass before the release of the tokens starts. After the cliff period is over, the vesting period will start. While examining the current vesting logic, we notice the implementation needs to be improved. to accommodate the cliff requirement.

To elaborate, we show below the related helper routine, i.e., _computeReleasableAmount(), which is used to compute the releasable amount from the vesting. Our analysis shows that the current computation either takes no consideration on the intended cliff or cannot release any initial amount before the cliff.

```
    function _computeReleasableAmount(
        VestingSchedule memory vestingSchedule
    ) internal view returns (uint256) {
        uint256 currentTime = getCurrentTime();
        if (
            (currentTime < vestingSchedule.start) ||
            vestingSchedule.revoked == true
        ) {
            return 0;
        } else if (
```

```
            currentTime >= vestingSchedule.start.add(vestingSchedule.duration)
        ) {
            return
                vestingSchedule.amountTotal.sub(vestingSchedule.amountReleased);
        } else if (
            vestingSchedule.amountReleased >= vestingSchedule.amountTotal
        ) {
            return 0;
        } else if (
            vestingSchedule.amountReleased < vestingSchedule.amountInitial
        ) {
            return
                vestingSchedule.amountInitial.sub(
                    vestingSchedule.amountReleased
                );
        } else {
            uint256 timeFromStart = currentTime.sub(vestingSchedule.start);
            uint secondsPerSlice = vestingSchedule.period;
            uint256 vestedSlicePeriods = timeFromStart.div(secondsPerSlice);
            uint256 vestedSeconds = vestedSlicePeriods.mul(secondsPerSlice);
            uint256 vestedAmount = vestingSchedule
                .amountTotal
                .mul(vestedSeconds)
                .div(vestingSchedule.duration);
            vestedAmount = vestedAmount.sub(vestingSchedule.amountReleased);

            return vestedAmount;
        }
    }
```

Listing 4.1: `RowaVesting::_computeReleasableAmount()`

Recommendation Take into account the cliff during the calculation of releasable amount from the vesting, including the intial release amount.

Status This issue has been resolved in the following commit: `a22118e`.

4.2 Improved Revoke Logic in RowaVesting

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RowaVesting`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

As mentioned earlier, the `RowaVesting` contract aims to instantiate different vesting schedules and allows beneficiaries to claim the vested tokens. Note that certain vesting schedules by design are allowed to be revoked. While examining the current revoke logic, we notice the current implementation can be improved.

To elaborate, we show below the related `revoke()` routine. As the name indicates, this routine allows the owner to revoke an active vesting schedule, if revokable. It comes to our attention the current revoke logic adjusts the various states, including `totalPSVested`, `totalPRIVSVested`, and `totalSEEDSVested`. However, both public sale and private sale are designed to be non-revocable. In other words, the related code logic becomes redundant and can be safely removed (lines 343-346).

```
321     function revoke(
322         bytes32 vestingScheduleId
323     ) public onlyOwner onlyActive(vestingScheduleId) {
324         VestingSchedule storage vestingSchedule = vestingSchedules[
325             vestingScheduleId
326         ];
327         require(
328             vestingSchedule.revokable == true,
329             "TokenVesting: vesting is not revocable"
330         );
331         uint256 vestedAmount = _computeReleasableAmount(vestingSchedule);
332         if (vestedAmount > 0) {
333             release(vestingScheduleId, vestedAmount);
334         }
335         uint256 unreleased = vestingSchedule.amountTotal.sub(
336             vestingSchedule.amountReleased
337         );
338 
339         vestingSchedulesTotalAmount = vestingSchedulesTotalAmount.sub(
340             unreleased
341         );
342 
343         if (equal(vestingSchedule.name, PS_VESTING_NAME)) {
344             totalPSVested = totalPSVested.sub(unreleased);
345         } else if (equal(vestingSchedule.name, PRIVS_VESTING_NAME)) {
346             totalPRIVSVested = totalPRIVSVested.sub(unreleased);
```

```
        } else if (equal(vestingSchedule.name, SEEDS_VESTING_NAME)) {
            totalSEEDSVested = totalSEEDSVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, TEAM_VESTING_NAME)) {
            totalTEAMVested = totalTEAMVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, ADVISORS_VESTING_NAME)) {
            totalADVISORSVested = totalADVISORSVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, PARTNERSHIPS_VESTING_NAME)) {
            totalPARTNERSHIPSVested = totalPARTNERSHIPSVested.sub(unreleased);
        }

        vestingSchedule.revoked = true;
    }
```

Listing 4.2: `RowaVesting::revoke()`

Recommendation Properly revise the above revoke logic to remove redundant processing.

Status This issue has been resolved in the following commit: `2ecddd5`.

4.3 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `RowaVesting`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `RowaVesting` implementation, there is a privileged accounts, i.e., `owner`. This account plays a critical role in governing and regulating the system-wide operations (e.g., add/revoke vesting schedules, withdraw tokens from the contract, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `RowaVesting` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    function revoke(
        bytes32 vestingScheduleId
    ) public onlyOwner onlyActive(vestingScheduleId) {
        VestingSchedule storage vestingSchedule = vestingSchedules[
            vestingScheduleId
        ];
        require(
            vestingSchedule.revokable == true,
            "TokenVesting: vesting is not revocable"
        );
        uint256 vestedAmount = _computeReleasableAmount(vestingSchedule);
        if (vestedAmount > 0) {
```

```
            release(vestingScheduleId, vestedAmount);
        }
        uint256 unreleased = vestingSchedule.amountTotal.sub(
            vestingSchedule.amountReleased
        );

        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount.sub(
            unreleased
        );

        if (equal(vestingSchedule.name, PS_VESTING_NAME)) {
            totalPSVested = totalPSVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, PRIVS_VESTING_NAME)) {
            totalPRIVSVested = totalPRIVSVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, SEEDS_VESTING_NAME)) {
            totalSEEDSVested = totalSEEDSVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, TEAM_VESTING_NAME)) {
            totalTEAMVested = totalTEAMVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, ADVISORS_VESTING_NAME)) {
            totalADVISORSVested = totalADVISORSVested.sub(unreleased);
        } else if (equal(vestingSchedule.name, PARTNERSHIPS_VESTING_NAME)) {
            totalPARTNERSHIPSVested = totalPARTNERSHIPSVested.sub(unreleased);
        }

        vestingSchedule.revoked = true;
    }

    /**
     * @notice Withdraw the specified amount if possible.
     * @param amount the amount to withdraw
     */
    function withdraw(uint256 amount) public nonReentrant onlyOwner {
        require(
            getWithdrawableAmount() >= amount,
            "TokenVesting: not enough withdrawable funds"
        );
        _token.safeTransfer(owner(), amount);
    }
```

Listing 4.3: Example Privileged Operations in `RowaVesting`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, 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 resolved in the following commit: `2ecddd5`.

5 Conclusion

In this security audit, we have examined the ROWA token design and implementation. 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 level vulnerabilities were discovered, we identified three issues that need to be promptly addressed. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.