# SMART CONTRACT AUDIT REPORT

for

# Parcel Payroll

Prepared By: Xiaomi Huang

PeckShield

March 25, 2023

## Document Properties

| | |
|---|---|
| Client | Parcel Payroll |
| Title | Smart Contract Audit Report |
| Target | Parcel Payroll |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Xiaotao Wu, Xuxian Jiang |
| Reviewed by | Patrick Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | March 25, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc | February 26, 2023 | Xuxian Jiang | Release Candidate #1 |

## 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 `Parcel Payroll` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About Parcel Payroll

`Parcel Payroll` protocol is designed for crypto payroll with the goal of building the infrastructure from ground up. The protocol utilizes funds stored in `Gnosis Safe multisig` with the `spending limit module` enabled for secure and flexible management of funds. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Parcel Payroll

| Item | Description |
|---:|---|
| Name | Parcel Payroll |
| Type | Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | March 25, 2023 |

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

- https://github.com/ParcelHQ/parcel-payroll.git (3fa78df)

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

- https://github.com/ParcelHQ/parcel-payroll.git (54dbcc9)

## 1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems 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).

Table 1.2: Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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 or analysis 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

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 |
| **Semantic Consistency Checks** | Semantic Consistency Checks |
| **Advanced DeFi Scrutiny** | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| **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 |

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- 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.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 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.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logics** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Parcel Payroll` protocol. 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 DeFi-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 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though 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 Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Revisited Logic in PayrollManager::executePayroll() | Business Logic | Resolved |
| PVE-002 | Low | Revisited Payout Nonce Retrieval/Update in PayrollManager | Coding Practices | Resolved |
| PVE-003 | Low | Improved Validation on User Input in PayrollManager::executePayroll() | Coding Practices | 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 3 for details.

# 3 Detailed Results

## 3.1 Revisited Logic in PayrollManager::executePayroll()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PayrollManager`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

### Description

As a payroll protocol, `Parcel Payroll` is designed to utilize funds stored in `Gnosis Safe multisig` with the `spending limit module` enabled for secure and flexible management of funds. While examining the built-in logic in handling the leftover native coins, we notice the current implementation needs to be improved.

In the following, we show below the specific routine, i.e., `executePayroll()`. As the name indicates, this routine is used to execute a specific payroll. By design, this contract will revert if there is any tokens left after the payroll execution. However, it comes to our attention that when validating whether there is any ether left, the current implementation enforces the following statement, i.e., `require(address(this).balance == initialBalances[i])` (line 235), which needs to be revised as `require(address(this).balance > initialBalances[i])`.

```
231         // Check if the contract has any tokens left
232         for (uint256 i = 0; i < paymentTokens.length; i++) {
233             if (paymentTokens[i] == address(0)) {
234                 // Revert if the contract has any ether left
235                 require(address(this).balance == initialBalances[i], "CS018");
236             } else if (
237                 IERC20(paymentTokens[i]).balanceOf(address(this)) >
238                 initialBalances[i]
239             ) {
240                 // Revert if the contract has any tokens left
241                 revert("CS018");
242             }
```

```
        }
```

Listing 3.1: `PayrollManager::executePayroll()`

**Recommendation** Revise the above `executePayroll()` routine to properly check whether there is any asset left.

**Status** This issue has been resolved as the initial balance check is removed.

## 3.2 Revisited Payout Nonce Retrieval/Update in PayrollManager

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PayrollManager`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

### Description

The audited protocol has a core `PayrollManager` contract, which makes use of a `packedPayoutNonces` array of `uint256` so that each `uint256` represents `256` payout nonces. While examining the related logic, we notice the payout nonce retrieval and update logic can be improved.

In current implementation, if we examine the related routine – `PayrollManager::packPayoutNonce()`, when the given slot is greater than the array length (lines 38-44), the first `if`-statement (line 38) is redundant as it is re-checked again in the next internal `while`-loop (lines 41-43).

```
24      function packPayoutNonce(
25          address safeAddress,
26          uint256 payoutNonce
27      ) internal {
28          // Packed payout nonces are stored in an array of uint256
29          // Each uint256 represents 256 payout nonces
30
31          // Each payout nonce is packed into a uint256, so the index of the uint256 in
                the array is the payout nonce / 256
32          uint256 slot = payoutNonce / 256;
33
34          // The bit index of the uint256 is the payout nonce % 256 (0-255)
35          uint256 bitIndex = payoutNonce % 256;
36
37          // If the slot is greater than the length of the array, we need to add more
                slots
38          if (orgs[safeAddress].packedPayoutNonces.length <= slot) {
39              // Add the required number of slots
```

```

            while (orgs[safeAddress].packedPayoutNonces.length <= slot) {
                orgs[safeAddress].packedPayoutNonces.push(0);
            }
        }

        // Set the bit to 1
        // This means that the payout nonce has been used
        orgs[safeAddress].packedPayoutNonces[slot] = 1 « bitIndex;
```

Listing 3.2: `PayrollManager::packPayoutNonce()`

A similar issue is also present in the `getPayoutNonce()` counterpart.

**Recommendation** Improve the current implementation in retrieving and updating the payout nonce in the above two routines.

**Status** This issue has been resolved as suggested.

## 3.3 Improved Validation on User Input in PayrollManager::executePayroll()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PayrollManager`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

### Description

The core `PayrollManager` contract is the main entry point with users and there is a need to properly validate the user input. Specifically, the `executePayroll()` routine can be improved to apply more rigorous input verification.

To elaborate, we show below the related implementation. This routine takes a number of arguments and it applies a number of same-length checks on the given array-related arguments, including `to`, `tokenAddress`, `amount`, and `payoutNonce`, as well as `paymentTokens` and `payoutAmounts`. It comes to our attention that the same-length check can also be applied on the given `proof`, which is currently missing.

```
    function executePayroll(
        address safeAddress,
        address[] memory to,
        address[] memory tokenAddress,
        uint128[] memory amount,
        uint64[] memory payoutNonce,
```

```
        bytes32[][][] memory proof,
        bytes32[] memory roots,
        bytes[] memory signatures,
        address[] memory paymentTokens,
        uint96[] memory payoutAmounts
    ) external nonReentrant whenNotPaused {
        // check if safe is onboarded
        require(orgs[safeAddress].approverCount != 0, "CS009");

        // Validate the Input Data
        require(to.length == tokenAddress.length, "CS004");
        require(to.length == amount.length, "CS004");
        require(to.length == payoutNonce.length, "CS004");
        require(roots.length == signatures.length, "CS004");
        require(paymentTokens.length == payoutAmounts.length, "CS004");
        ...
    }
```

Listing 3.3: `PayrollManager::executePayroll()`

**Recommendation** Strengthen the above `executePayroll()` routine to ensure all input arguments are properly validated

**Status** This issue has been confirmed.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the `Parcel Payroll` protocol, which is designed for crypto payroll with the goal of building the infrastructure from ground up. The protocol utilizes funds stored in `Gnosis Safe multisig` with the `spending limit module` enabled for secure and flexible management of funds. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[3] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.