SMART CONTRACT AUDIT REPORT

for

Protocolink

Prepared By: Xiaomi Huang

PeckShield

November 15, 2023

Document Properties

Client	Protocolink
Title	Smart Contract Audit Report
Target	Protocolink
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Colin Zhong
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 15, 2023	Xuxian Jiang	Final Release
1.0-rc	November 11, 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 smart contract source code of the `Protocolink` protocol, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About Protocolink

`Protocolink` is a router system which consolidates protocol interactions within a secure `Router/Agent` architecture, facilitating single-transaction processes `ERC-20`, `ERC-721`, `ERC-1155` and lending positions. Its contracts are protocol-agnostic, with all protocol-specific code externalized, thus enhancing flexibility and immutable contracts. For developers, Protocolink provides a comprehensive `API` and `SDK` designed for the creation of complex transactions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Protocolink

Item	Description
Name	Protocolink
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 15, 2023

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

- https://github.com/dinngo/protocolink-contract.git (3ebf8bb)

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

- https://github.com/dinngo/protocolink-contract.git (1c7d566)

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 classified into four categories accordingly, 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

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

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.

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.

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 Logic	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 implementation of the `Protocolink` 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	0	
Low	2	■ ■
Informational	1	■
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 2 low-severity vulnerabilities and 1 informational issue.

Table 2.1: Key Protocolink Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved _executeLogics() Handling in AgentImplementation	Coding Practices	Resolved
PVE-002	Informational	Revisited Allowance Management in AgentImplementation	Coding Practices	Resolved
PVE-003	Low	Trust Issue Of Admin Keys	Security Features	Resolved

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 Improved _executeLogics() Handling in AgentImplementation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AgentImplementation
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The Protocolink protocol is designed to be protocol-agnostic, with all protocol-specific code externalized. While examining the interaction with external protocols, we notice a minor issue that may improve current key `_executeLogics()` routine.

Specifically, we show below the code snippet from the `_executeLogics()` routine. It has a rather straightforward logic in calculating native or token amount. By design, it supports the direct replacement of the token amount in the input data. The direct replacement will be indicated with `balanceBps!=_BPS_NOT_USED` and a valid `offset` for replacement. We suggest to strengthen the validation of the `offset` by adding the following requirement: `require(offset + 0x24 <= data.length)`.

```
            // Calculate native or token amount
            // 1. if balanceBps is `_BPS_NOT_USED`, then `amountOrOffset` is
                interpreted directly as the amount.
            // 2. if balanceBps isn't `_BPS_NOT_USED`, then the amount is calculated
                by the balance with bps
            uint256 amount;
            if (balanceBps == _BPS_NOT_USED) {
                amount = inputs[j].amountOrOffset;
            } else {
                if (balanceBps > _BPS_BASE) revert InvalidBps();

                if (token == wrappedNative && wrapMode == DataType.WrapMode.
                    WRAP_BEFORE) {
```

```
                // Use the native balance for amount calculation as wrap will be executed later
                amount = (address(this).balance * balanceBps) / _BPS_BASE;
            } else {
                amount = (_getBalance(token) * balanceBps) / _BPS_BASE;
            }

            // Check if the calculated amount should replace the data at the offset. For most protocols that use
            // `msg.value` to pass the native amount, use `_OFFSET_NOT_USED` to indicate no replacement.
            uint256 offset = inputs[j].amountOrOffset;
            if (offset != _OFFSET_NOT_USED) {
                assembly {
                    let loc := add(add(data, 0x24), offset) // 0x24 = 0x20(data_length) + 0x4(sig)
                    mstore(loc, amount)
                }
            }
            emit AmountReplaced(i, j, amount);
        }
```

Listing 3.1: AgentImplementation::_executeLogics()

Recommendation Improve the above routine by further validating the given `offset` for the in-place replacement.

Status The issue has been fixed by this commit: `1c7d566`.

3.2 Revisited Allowance Management in AgentImplementation

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AgentImplementation`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, `Protocolink` is a router system that effectively consolidates protocol interactions within a secure `Router/Agent` architecture. While analyzing the fund movement in this architecture, we notice the need to reset possible token allowance that may be previously granted to external contracts.

To elaborate, we show below the related code snippet from the `_executeLogics()` routine. The token approval is necessary to prepare the input tokens before interacting with external protocols.

However, it comes to our attention that the granted token approvals may be unlimited (line 251) and is suggested to revoke right after the external interaction.

```
211             for (uint256 j; j < inputsLength; ) {
212                 address token = inputs[j].token;
213                 uint256 balanceBps = inputs[j].balanceBps;
214 
215                 // Calculate native or token amount
216                 // 1. if balanceBps is '_BPS_NOT_USED', then 'amountOrOffset' is
                        interpreted directly as the amount.
217                 // 2. if balanceBps isn't '_BPS_NOT_USED', then the amount is calculated
                         by the balance with bps
218                 uint256 amount;
219                 if (balanceBps == _BPS_NOT_USED) {
220                     amount = inputs[j].amountOrOffset;
221                 } else {
222                     if (balanceBps > _BPS_BASE) revert InvalidBps();
223 
224                     if (token == wrappedNative && wrapMode == DataType.WrapMode.
                            WRAP_BEFORE) {
225                         // Use the native balance for amount calculation as wrap will be
                                 executed later
226                         amount = (address(this).balance * balanceBps) / _BPS_BASE;
227                     } else {
228                         amount = (_getBalance(token) * balanceBps) / _BPS_BASE;
229                     }
230 
231                     // Check if the calculated amount should replace the data at the
                            offset. For most protocols that use
232                     // 'msg.value' to pass the native amount, use '_OFFSET_NOT_USED' to
                            indicate no replacement.
233                     uint256 offset = inputs[j].amountOrOffset;
234                     if (offset != _OFFSET_NOT_USED) {
235                         assembly {
236                             let loc := add(add(data, 0x24), offset) // 0x24 = 0x20(
                                    data_length) + 0x4(sig)
237                             mstore(loc, amount)
238                         }
239                     }
240                     emit AmountReplaced(i, j, amount);
241                 }
242 
243                 if (token == wrappedNative && wrapMode == DataType.WrapMode.WRAP_BEFORE)
                     {
244                     // Use += to accumulate amounts with multiple WRAP_BEFORE, although
                            such cases are rare
245                     wrappedAmount += amount;
246                 }
247 
248                 if (token == _NATIVE) {
249                     value += amount;
250                 } else if (token != approveTo) {
```

```
                ApproveHelper.approveMax(token, approveTo, amount);
            }

            unchecked {
                ++j;
            }
        }
```

Listing 3.2: `AgentImplementation::_executeLogics()`

Recommendation Revise the above routine to revoke previously-approved allowance after the interaction with external protocols.

Status The issue has been resolved as the `agent` by design is not expected to hold assets.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Protocolink` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., add/remove signers and configure various parameters). In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    /// @param amount The amount of tokens to be rescued
    function rescue(address token, address receiver, uint256 amount) external onlyOwner
        {
        IERC20(token).safeTransfer(receiver, amount);
    }

    /// @notice Add a signer whose signature can pass the validation in `
        executeWithSignerFee` by owner
    /// @param signer The signer address to be added
    function addSigner(address signer) external onlyOwner {
        signers[signer] = true;
        emit SignerAdded(signer);
    }

    /// @notice Remove a signer by owner
    /// @param signer The signer address to be removed
    function removeSigner(address signer) external onlyOwner {
```

```
        delete signers[signer];
        emit SignerRemoved(signer);
    }

    /// @notice Set a new fee rate by owner
    /// @param feeRate_ The new fee rate in basis points
    function setFeeRate(uint256 feeRate_) external onlyOwner {
        if (feeRate_ >= _BPS_BASE) revert InvalidRate();
        feeRate = feeRate_;
        emit FeeRateSet(feeRate_);
    }
```

Listing 3.3: Example Privileged Operations in `Router`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these `onlyOwner` privileges explicit or raising necessary awareness among protocol users.

Recommendation Making these `onlyOwner` privileges explicit among protocol users.

Status This issue has been mitigated. And the team clarifies that the owner cannot proactively attack the various approvals users have on the `Agent`, nor can owner upgrade the contract. This approach is intended to enhance the security of the `Protocolink` contracts.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Protocolink` protocol, which is a router system which consolidates protocol interactions within a secure `Router/Agent` architecture, facilitating single-transaction processes `ERC-20`, `ERC-721`, `ERC-1155` and lending positions. Its contracts are protocol-agnostic, with all protocol-specific code externalized, thus enhancing flexibility and immutable contracts. For developers, Protocolink provides a comprehensive `API` and `SDK` designed for the creation of complex transactions. The current code base is well organized and 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-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] 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.