SMART CONTRACT AUDIT REPORT

for

ZigZag Forwarder

Prepared By: Xiaomi Huang

PeckShield
November 24, 2023

Document Properties

Client	ZigZag
Title	Smart Contract Audit Report
Target	Forwarder
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 24, 2023	Xuxian Jiang	Final Release
1.0-rc1	November 21, 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 `Forwarder` 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 Forwarder

`Forwarder` contract is a mix between `EIP2771 meta transactions` (gasless) and session wallets. It is used to set allowed signers and forward those requests to the target contract. It also has a related `Fee collector` that converts fees to the fee token and another `ChainSpecificUtil` contract for the calculation of the `L1` fee part on `Arbitrum`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Forwarder

Item	Description
Name	ZigZag
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 24, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers the following three contracts: `ChainSpecificUtil.sol`, `FeeCollector.sol`, and `Forwarder.sol`.

- https://github.com/ZigZagExchange/Forwarder.git (5b15330)

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

- https://github.com/ZigZagExchange/Forwarder.git (422cd2b)

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

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 contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3:   The Full List of Check Items

Category	Check Item
	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
Basic Coding Bugs	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
	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
Advanced DeFi Scrutiny	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
Additional Recommendations	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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 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 implementation of the `Forwarder` 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	0	
Total	2	

We have so far identified a list of potential issues. 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 that 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.

Table 2.1:   Key Forwarder Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, 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 details.

3 | Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `FeeCollector`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()`/`transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194    /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
             of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201        // To change the approve amount you first have to reduce the addresses'
202        //  allowance to zero by calling 'approve(_spender, 0)' if it is not
203        //  already 0 to mitigate the race condition described here:
204        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
207          allowed[msg.sender][_spender] = _value;
208          Approval(msg.sender, _spender, _value);
209      }
```

<p align="center">Listing 3.1: USDT Token **Contract**</p>

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
38      /**
39       * @dev Deprecated. This function has issues similar to the ones found in
40       * {IERC20-approve}, and its usage is discouraged.
41       *
42       * Whenever possible, use {safeIncreaseAllowance} and
43       * {safeDecreaseAllowance} instead.
44       */
45      function safeApprove(
46          IERC20 token,
47          address spender,
48          uint256 value
49      ) internal {
50          // safeApprove should only be called when setting an initial allowance,
51          // or when resetting it to zero. To increase and decrease it, use
52          // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53          require(
54              (value == 0)  (token.allowance(address(this), spender) == 0),
55              "SafeERC20: approve from non-zero to non-zero allowance"
56          );
57          _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
                 spender, value));
58      }
```

<p align="center">Listing 3.2: `SafeERC20::safeApprove()`</p>

In current implementation, if we examine the `FeeCollector::collectFees()` routine that is designed to collect the protocol fees. To accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()`, instead of `transferFrom()` (line 149).

```
142   function collectFees(address requestedFeeTokenAddress, address requestSender, uint256
          gasUsed, uint256 valueUsed) public {
143     if (forwarderAddress != _msgSender()) revert ForwarderUnauthorizedAccount(
          forwarderAddress, _msgSender());
144     if (address(feeToken) != requestedFeeTokenAddress) revert WrongFeeToken(address(
          feeToken), requestedFeeTokenAddress);
145     if (minGasUsed > gasUsed) revert LessThenMinimumGas(minGasUsed, gasUsed);
146
147     uint256 amount = estimateFee(valueUsed, gasUsed);
```

```
148
149      bool success = ERC20(feeToken).transferFrom(requestSender, paymentAddress, amount);
150      if (!success) revert FailedToCollectFee();
151
152      emit FeesCollected(requestSender, paymentAddress, amount, gasUsed);
153    }
```

Listing 3.3: `FeeCollector::collectFees()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transferFrom()`.

Status This issue has been fixed in the following PR: 4.

3.2 Trust Issue of Admin Keys

- ID: PVE-002

- Severity: Low

- Likelihood: Low

- Impact: Low

- Target: `Forwarder`

- Category: Security Features [3]

- CWE subcategory: CWE-287 [2]

Description

In `Forwarder`, there is a privileged administrative account (`owner`). The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Forwarder` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
277    function addFeeCollectorAddress(address feeTokenAddress, address feeCollectorAddress)
           public onlyOwner {
278      require(feeTokenAddress != address(0), "Forwarder: Can not set to address zero");
279
280      feeCollectorMapping[feeTokenAddress] = IFeeCollector(feeCollectorAddress);
281      emit AddedFeeCollector(feeTokenAddress);
282    }
283
284    /// @notice Remove a fee collector
285    /// @dev This will set the fee collector address for that ERC20 token to zero
286    /// @param feeTokenAddress ERC20 token address
287    function removeFeeCollectorAddress(address feeTokenAddress) public onlyOwner {
288      require(feeTokenAddress != address(0), "Forwarder: Can not set to address zero");
289
290      feeCollectorMapping[feeTokenAddress] = IFeeCollector(address(0));
291      emit RemovedFeeCollector(feeTokenAddress);
292    }
```

```
293
294   /// @notice pause forwarder
295   /// @dev This will stop the contract and not allow any further execute request calls
296   function pause() public onlyOwner {
297     _pause();
298   }
299
300   /// @notice unpaus forwarder
301   /// @dev This will resume the contract and allow new execute request calls
302   function unpause() public onlyOwner {
303     _unpause();
304   }
```

Listing 3.4: Example Privileged Operations in `Forwarder`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative 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 mitigated with the plan to transfer the privileged account to a `multi-sig` account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Forwarder` protocol, which is a mix between `EIP2771 meta transactions` (gasless) and session wallets. It is used to set allowed signers and forward those requests to the target contract. It also has a related `Fee collector` that converts fees to the fee token and another `ChainSpecificUtil` contract for the calculation of the `L1` fee part on `Arbitrum`. 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-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.