SMART CONTRACT AUDIT REPORT

for

UXLINK ERC20 MultiSender

Prepared By: Xiaomi Huang

PeckShield

September 27, 2025

Document Properties

Client	UXLINK
Title	Smart Contract Audit Report
Target	UXLINK ERC20 MultiSender
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 27, 2025	Xuxian Jiang	Final Release
1.0-rc	September 26, 2025	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 ERC20 MultiSender contract, 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 ERC20 MultiSender

ERC20 MultiSender, alternatively called a multi-transfer or bulk sender, is a smart contract or tool that allows to send ERC20 tokens to many addresses in a single transaction, instead of sending tokens one by one. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of UXLINK ERC20 MultiSender

Item	Description
Name	UXLINK
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 27, 2025

In the following, we show the deployment address of the audited contract.

- https://sepolia.etherscan.io/address/0xc45B7f627feF5979760aECF10bC73065dD250FF8

And this is the new deployment address after all fixes for the issues found in the audit have been checked in:

- https://sepolia.etherscan.io/address/0x7D10deDe472f482dE67227e3f83DBf574F5F9347

1.2 About PeckShield

PeckShield Inc. [5] 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 [4]:

- 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) [3], 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 implementation of the `ERC20 MultiSender` smart 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 low-severity vulnerability and 1 informational issue.

Table 2.1: Key Audit Findings in UXLINK ERC20 MultiSender

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Resolved
PVE-002	Informational	Improved Gas Efficiency in Batch Transfers	Coding Practice	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 contract is being deployed on 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: UXLINKERC20MultiSender
- Category: Coding Practice [2]
- CWE subcategory: CWE-563 [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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
```

```
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.1: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `batch_transfer()` routine in the `UXLINKERC20MultiSender` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transferFrom(msg.sender, address(this), fee)` (lines 127 and 129) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
121     function batch_transfer(address _token, address[] memory to, uint256 amount) public {
122         IERC20 token = IERC20(_token);
123         uint256 fee = calculateWithdrawalFee(amount);
124         uint256 amountAfterFee = amount - fee;
125         for (uint256 i = 0; i < to.length; i++) {
126             if (fee > 0) {
127                 token.transferFrom(msg.sender, address(this), fee);
128             }
129             token.transferFrom(msg.sender, to[i], amountAfterFee);
130         }
131     }
```

Listing 3.2: `UXLINKERC20MultiSender::batch_transfer()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note another function `batch_transfer_diffent_amount()` shares the same issue.

Status This issue has been resolved by following the above suggestion.

3.2 Improved Gas Efficiency in Batch Transfers

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UXLINKERC20MultiSender`
- Category: Coding Practice [2]
- CWE subcategory: CWE-563 [1]

Description

The bulk-sending feature of the audited multi-sender contract is convenient and gas-friendly when compared to the need of sending separate transactions for each recipient. While reviewing the current implementation, we notice an opportunity to further reduce gas cost.

In the following, we show the implementation of a related function `batch_transfer()`. By design, the batch transfer feature allows to charge certain withdrawal fee. Our analysis shows that the withdraw fee is collected from the calling user to the contract itself multiple times (line 127), not once. As a result, we can revise it by calculating the total withdraw fee and collecting the fee once.

```
121     function batch_transfer(address _token, address[] memory to, uint256 amount) public {
122         IERC20 token = IERC20(_token);
123         uint256 fee = calculateWithdrawalFee(amount);
124         uint256 amountAfterFee = amount - fee;
125         for (uint256 i = 0; i < to.length; i++) {
126             if (fee > 0) {
127                 token.transferFrom(msg.sender, address(this), fee);
128             }
129             token.transferFrom(msg.sender, to[i], amountAfterFee);
130         }
131     }
```

Listing 3.3: `UXLINKERC20MultiSender::batch_transfer()`

Recommendation Revise the above logic to efficiently collect withdraw fee. Note the same suggestion is also applicable to the `batch_transfer_diffent_amount()` routine. Their revisions are show below:

```
function batch_transfer(address _token, address[] memory to, uint256 amount) public {
    IERC20 token = IERC20(_token);
    uint256 fee = calculateWithdrawalFee(amount);
    uint256 amountAfterFee = amount - fee;
    for (uint256 i = 0; i < to.length; i++) {
        token.transferFrom(msg.sender, to[i], amountAfterFee);
    }

```

```
        if (fee > 0) {
            token.transferFrom(msg.sender, address(this), fee * to.length);
        }

    }
```

Listing 3.4: Revised `UXLINKERC20MultiSender::batch_transfer()`

```
    function batch_transfer_diffent_amount(address _token, address[] memory to, uint[]
        memory amount) public {
        IERC20 token = IERC20(_token);
        require(to.length == amount.length, "address.len must equal amount.len ");

        uint256 total_fee;
        for (uint256 i = 0; i < to.length; i++) {
            uint256 fee = calculateWithdrawalFee(amount[i]);
            uint256 amountAfterFee = amount[i] - fee;

            total_fee += fee;
            token.safeTransferFrom(msg.sender, to[i], amountAfterFee);
        }

        if (total_fee > 0) {
            token.safeTransferFrom(msg.sender, address(this), total_fee);
        }

    }
```

Listing 3.5: Revised `UXLINKERC20MultiSender::batch_transfer_diffent_amount()`

Status This issue has been resolved by following the above suggestion.

4 Conclusion

In this audit, we have analyzed the design and implementation of the ERC20 MultiSender contract, which is used to batch transfer ERC20 tokens to multiple recipient addresses in a single transaction, instead of sending tokens one by one. 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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