SMART CONTRACT AUDIT REPORT

for

UXLINK Debit Card

Prepared By: Xiaomi Huang

PeckShield
April 30, 2025

Document Properties

Client	UXLINK
Title	Smart Contract Audit Report
Target	UXLINK Debit Card
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xuxian Jiang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 30, 2025	Xuxian Jiang	Final Release
1.0-rc	April 29, 2025	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 UXLINK Debit Card smart 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 UXLINK Debit Card

The UXLINKDebitCard is an Ethereum-based smart contract protocol that serves as a wallet provider interface for the Fiat24 ecosystem. This protocol enables users to mint Fiat24 NFTs and facilitates token deposits and transfers. The basic information of the audited contract is as follows:

Table 1.1: Basic Information of The UXLINK Debit Card Contract

Item	Description
Name	UXLINK Debit Card
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 30, 2025

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

- https://arbiscan.io/address/0x85BaCa36C0DC02a6b935b9a3816860ceDf7077b5

And here is the new deployment address of the audited contract after all fixes have been checked in:

- https://arbiscan.io/address/0x56bF220e65836A54f4835d35d8D2bA2feF7e4587

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 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.

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 `UXLINK Debit Card` implementation. 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 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 and 1 informational recommendation.

Table 2.1: Key UXLINK Debit Card Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited depositForClient() Logic in UXLINKDebitCard	Business Logic	Resolved
PVE-002	Informational	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-003	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 Revisited depositForClient() Logic in UXLINKDebitCard

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UXLINKDebitCard
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The UXLINKDebitCard contract has a core function that allows to deposit USDC into the configured FIAT24_CRYPTO_DEPOSIT on behalf of a given client. While reviewing current logic in setting the token allownace, we notice an issue that may not be consistent with the FIAT24_CRYPTO_DEPOSIT logic.

To elaborate, we show below the implementation of the related depositForClient() routine as well as the associated depositByWallet() routine. We notice the USDC token spending allowance is configured on the UXLINKDebitCard side while the actual transfer occurs on the FIAT24_CRYPTO_DEPOSIT side with funds sourced from the given _client (lines 355 – 356), not the calling contract, i.e., FIAT24_CRYPTO_DEPOSIT.

```
    function depositForClient(
        address _client,
        address _outputToken,
        uint256 _usdcAmount
    ) external onlyManager nonReentrant {
        require(_isValidOutputToken(_outputToken), "Invalid output token");
        require(_usdcAmount > 0, "Zero amount not allowed");
        require(USDC.balanceOf(address(this)) >= _usdcAmount, "Insufficient USDC balance
            ");

        // Approve USDC for Fiat24CryptoDeposit contract
        USDC.approve(address(FIAT24_CRYPTO_DEPOSIT), _usdcAmount);

        FIAT24_CRYPTO_DEPOSIT.depositByWallet(_client, _outputToken, _usdcAmount);
```

```
        emit DepositForClient(_client, address(USDC), _outputToken, _usdcAmount);
    }
```

Listing 3.1: `UXLINKDebitCard::depositForClient()`

```
    function depositByWallet(address _client, address _outputToken, uint256 _usdcAmount) external returns(uint256) {
        if(paused()) revert Fiat24CryptoDeposit__Paused();
        if(_usdcAmount < minUsdcDepositAmount) revert Fiat24CryptoDeposit__UsdcAmountLowerMinDepositAmount(_usdcAmount, minUsdcDepositAmount);
        uint256 tokenId = IFiat24Account(fiat24account).tokenOfOwnerByIndex(_client, 0);
        if(IFiat24Account(fiat24account).walletProvider(tokenId) != IFiat24Account(fiat24account).tokenOfOwnerByIndex(_msgSender(), 0)) {
            revert Fiat24CryptoDeposit__NotTokensWalletProvider(_msgSender(), tokenId);
        }

        uint256 feeInUSDC = getFee(tokenId, _usdcAmount);
        uint256 outputAmount = (_usdcAmount - feeInUSDC) / USDC_DIVISOR * exchangeRates[usdc][usd24] / XXX24_DIVISOR;
        outputAmount = outputAmount * getExchangeRate(usd24, _outputToken) / XXX24_DIVISOR * getSpread(usd24, _outputToken,false) / XXX24_DIVISOR;
        TransferHelper.safeTransferFrom(usdc, _client, usdcDepositAddress, _usdcAmount - feeInUSDC);
        TransferHelper.safeTransferFrom(usdc, _client, _msgSender(), feeInUSDC);
        TransferHelper.safeTransferFrom(_outputToken, IFiat24Account(fiat24account).ownerOf(CRYPTO_DESK), _client, outputAmount);

        emit DepositedByWallet(tokenId,
                               _client,
                               IFiat24Account(fiat24account).tokenOfOwnerByIndex(_msgSender(), 0),
                               _msgSender(),
                               _outputToken,
                               _usdcAmount);

        return outputAmount;
    }
```

Listing 3.2: `FIAT24_CRYPTO_DEPOSIT::depositByWallet()`

Recommendation Properly revise the above routines to resolve possible inconsistency in sourcing the `USDC` funds for the deposit.

Status The issue has been resolved by removing the unwanted token allowance.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UXLINKDebitCard`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

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 3.3: ZRX::transfer()/transferFrom()

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 `withdrawToken()` routine in the `UXLINKDebitCard` contract. If the `USDT` token is provided as `_token`, the unsafe version of `IERC20(_token).transfer(_recipient, _amount);` (line 157) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
149     function withdrawToken(
150         address _token,
151         uint256 _amount,
152         address _recipient
153     ) external onlyManager nonReentrant {
154         require(_amount > 0, "Zero amount not allowed");
155         require(IERC20(_token).balanceOf(address(this)) >= _amount, "Insufficient balance");
156 
157         IERC20(_token).transfer(_recipient, _amount);
158 
159         emit TokenWithdrawn(_token, _amount, _recipient);
160     }
```

Listing 3.4: `UXLINKDebitCard::withdrawToken()`

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

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

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `UXLINKDebitCard`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `UXLINKDebitCard` contract, there is a privileged account, i.e., `manager`, which plays a critical role in governing and regulating the staking-wide operations (e.g., configure parameters and add new managers). It also has the privilege to affect the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function depositForClient(
        address _client,
        address _outputToken,
        uint256 _usdcAmount
    ) external onlyManager nonReentrant {...}

    ...
    function updateProviderConfig(
        uint256 _f24Required,
        address _feeToken,
        uint256 _mintFee
    ) external onlyManager {...}

    ...
    function withdrawToken(
        address _token,
        uint256 _amount,
        address _recipient
    ) external onlyManager nonReentrant {...}

    ...
    function withdrawETH(
        uint256 _amount,
        address _recipient
    ) external onlyManager nonReentrant {...}

    ...
    function transferContractNFT(
        address _nftContract,
        uint256 _tokenId,
        address _to
    ) external onlyManager nonReentrant {...}
```

Listing 3.5: Example Privileged Operations in the `UXLINKDebitCard` Contract

If the privileged admins are managed by a plain `EOA` account, this may be worrisome and pose counter-party risk to the exchange users. 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`. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contract has the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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 a multi-sig account to take the role of the manager.

4 Conclusion

In this audit, we have analyzed the design and implementation of `UXLINKDebitCard`, which is an `Ethereum`-based smart contract protocol and serves as a wallet provider interface for the `Fiat24` ecosystem. This protocol enables users to mint `Fiat24 NFTs` and facilitates token deposits and transfers. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.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.