SMART CONTRACT AUDIT REPORT

for

UXWallet Account

Prepared By: Xiaomi Huang

PeckShield

July 29, 2025

Document Properties

Client	UXLINK
Title	Smart Contract Audit Report
Target	UXWallet
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	July 29, 2025	Xuxian Jiang	Final Release
1.0-rc	July 27, 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 `UXWallet` account, 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 UXWallet

The `UXWallet` account is temporarily upgraded to a smart contract account via the `EIP-7702` standard, enabling advanced account abstraction features such as batch transactions, auto or sponsored gas payments, custom verification logic, social recovery mechanisms, and phishing protection. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of UXWallet

Item	Description
Name	UXLINK
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	July 29, 2025

In the following, we show the deployment addresses of the audited `UXWallet` wallet contracts.

- https://sepolia.arbiscan.io/address/0x80fF0245e913bCb6bce5E2488472Af7de9534327
- https://sepolia.arbiscan.io/address/0x31D5752909a1b50B2C9018B190Cb078D0d712df9
- https://sepolia.arbiscan.io/address/0x6A1a7d0873a82Bc45B5b7d3Df41604cE310735dD

And here are the new deployment addresses after all fixes for the issues found in the audit have been checked in:

- https://sepolia.arbiscan.io/address/0x4701D377dB334927E928498981622BCe6C4E881D
- https://sepolia.arbiscan.io/address/0x469555D052Ad53CA0C032cDb22C32964476D0355
- https://sepolia.arbiscan.io/address/0x42b9DFdbFd36d2289FaC6cd46512AF2eE6DeA07F

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 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) [9], 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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `UXWallet` wallet. 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	3	🟦 🟦 🟦
Informational	0	
Total	4	

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 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key UXWallet Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor Logic in Proxy Contracts	Coding Practices	Resolved
PVE-002	Low	Possible Denial-Of-Service in Signature Replay in UX7702ValidatorProxy	Time And State	Resolved
PVE-003	Low	Improved Validation of Function Arguments	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved Constructor Logic in Proxy Contracts

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

- Target: UX7702ValidatorProxy
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, the UX7702ValidatorProxy constract is instantiated as a proxy with actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., _disableInitializers ();. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the initializer() function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
contract UX7702ValidatorProxy is Initializable, UUPSUpgradeable,
    ReentrancyGuardUpgradeable, OwnableUpgradeable {
  using ECDSA for bytes32;
  uint256 public chainId;
  address private signer;
  mapping(address account => uint256 nonce) private nonces;
  mapping(bytes32 => uint256) private sigHashToRandom;

  function initialize(uint256 _chainId) public initializer {
    __Ownable_init();
    __ReentrancyGuard_init();
    chainId = _chainId;
    signer = msg.sender;
  }
  ...
}
```

Listing 3.1: `UX7702ValidatorProxy::initialize()`

Note the above `initializer()` function can also be improved by also invoking `__UUPSUpgradeable_init()` to initialize one of its parent contracts, i.e., `UUPSUpgradeable`.

Recommendation Improve the above-mentioned initialize routine in `UX7702ValidatorProxy`.

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

3.2 Possible Denial-Of-Service in Signature Replay in UX7702ValidatorProxy

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `UX7702ValidatorProxy`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The `UXWallet` account is `EIP-7702`-compliant and enables a variety of advanced account abstraction features. In the process of analyzing the signature validation logic in user authorization, we notice a frontruning risk that may introduce a denial-of-service issue.

```
  function verifySign(address caller, bytes calldata signature, address[] calldata
      targets, uint256[] calldata values, bytes[] calldata datas, uint256 random)
      external nonReentrant {
    require(sigHashToRandom[keccak256(signature)] == 0, "Exists signature");
    sigHashToRandom[keccak256(abi.encodePacked(signature, caller))] = random;
    uint256 nonce = _checkSignature(caller, signature, targets, values, datas);
    require(nonce > nonces[caller], "Invalid nonce");
    nonces[caller] = nonce;
  }
```

Listing 3.2: `UX7702ValidatorProxy::verifySign()`

To elaborate, we show above the implementation of the related `verifySign()` routine. As the name indicates, this routine verifies a given user signature. If the given signature is successfully validated, the caller account's nonce is updated with the enclosed one. However, before the `uxExecuteBatch()` transaction (that invokes the `verifySign()` call) is mined or included on the blockchain, it may be broadcasted and publicly seen. With that, a malicious user may extract the related signatures, targets, values, datas, and front-run to call `verifySign()`. By doing so, the caller's nonce maintained in `UX7702ValidatorProxy` is advanced to the enclosed nonce. When the observed `uxExecuteBatch()` transaction executes, it will be reverted since the nonce is already used! A mitigating approach will be imposing addition validation on the calling user (or caller) and we will elaborate it in Section 3.3.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of wallet users.

Status This issue has been resolved by ensuring the given caller argument is identical to the calling user, i.e., `msg.user`.

3.3 Improved Validation of Function Arguments

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UX7702Delegator`, `UX7702ValidatorProxy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `UXWallet` wallet follows the `EIP-7702` standard and enables a variety of advanced account abstraction features. While reviewing the lazy-minting logic, we notice the logic of delegating an `EOA` to a contract, we notice the input arguments can be benefited from improved validation.

To elaborate, we show below the implementation of a related `_checkSignature()` routine. This routine is used to validate the given signature is indeed signed by the trusted signer. It comes to our attention that it validates the given `signature` has at least `65` bytes (line 54), which should be improved by revising it to `65 + 128` (lines 55 – 56).

```
53  function _checkSignature(address caller, bytes calldata signature, address[] memory
        targets, uint256[] memory values, bytes[] memory datas) internal view returns (
        uint256) {
54    require(signature.length >= 65, "Signature too short");
55    (uint256 sigChainId, uint256 validUntil, uint256 validAfter, uint256 nonce) = abi.
          decode(signature[0:128], (uint256, uint256, uint256, uint256));
56    bytes memory signatureData = signature[signature.length - 65:];
57    require(sigChainId == chainId, "Invalid chainId");
58    require(block.timestamp >= validAfter, "Not valid yet");
59    require(block.timestamp <= validUntil, "Expired");
60    bytes32 hash = ECDSA.toEthSignedMessageHash(_toMessageHash(sigChainId, validUntil,
          validAfter, nonce, caller, targets, values, datas));
61    address addr = ECDSA.recover(hash, signatureData);
62    require(signer == addr, "Invalid signer");
```

```
    return nonce;
  }
```

Listing 3.3: `UX7702ValidatorProxy::_checkSignature()`

Moreover, the `checkSign()` function in `UX7702ValidatorProxy` indicates the given `caller` argument must match with the calling user (line 43). With that, we shall enforce the same requirement in `UX7702Delegator::uxExecuteBatch()`. This may not only perform the early validation, but simplify the logic and address the issue in Section 3.2.

```
42    function checkSign(address caller, bytes calldata signature, uint256 random) external
          {
43      require(msg.sender == caller, "Not caller");
44      bytes32 sigHash = keccak256(abi.encodePacked(signature, caller));
45      require(sigHashToRandom[sigHash] == random, "Signature check error");
46      delete sigHashToRandom[sigHash];
47    }
```

Listing 3.4: `UX7702ValidatorProxy::checkSign()`

Last but not least, the `verifySign()` function in `UX7702ValidatorProxy` may be improved by enforcing the following requirement, i.e., `require(sigHashToRandom[keccak256(abi.encodePacked(signature, caller))] == 0`, not current `require(sigHashToRandom[keccak256(signature)] == 0` (line 35).

```
34    function verifySign(address caller, bytes calldata signature, address[] calldata
          targets, uint256[] calldata values, bytes[] calldata datas, uint256 random)
          external nonReentrant {
35      require(sigHashToRandom[keccak256(signature)] == 0, "Exists signature");
36      sigHashToRandom[keccak256(abi.encodePacked(signature, caller))] = random;
37      uint256 nonce = _checkSignature(caller, signature, targets, values, datas);
38      require(nonce > nonces[caller], "Invalid nonce");
39      nonces[caller] = nonce;
40    }
```

Listing 3.5: `UX7702ValidatorProxy::verifySign()`

Recommendation Revisit the above-mentioned routines to ensure the given user input arguments are properly validated.

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

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: UX7702ValidatorProxy
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the UXWallet account, there is a privileged account owner that plays a critical role in governing and regulating the system-wide operations (e.g., manger the signer, configure the chainId parameter, and upgrade the proxy contract). The account also has the privilege to control or govern 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 the related privileged accesses in current contracts.

```
function _authorizeUpgrade(address newImplementation) internal override onlyOwner {}

function setChainId(uint256 _chainId) external onlyOwner {
  chainId = _chainId;
}

function setSigner(address _addr) external onlyOwner {
  signer = _addr;
}
```

Listing 3.6: Example Privileged Functions in UX7702ValidatorProxy

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts 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 privileges explicit or raising necessary awareness among protocol users.

In the meantime, the protocol contracts are deployed as proxies to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

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 as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the UXWallet account system, which is temporarily upgraded to a smart contract account via the EIP-7702 standard, enabling advanced account abstraction features such as batch transactions, auto or sponsored gas payments, custom verification logic, social recovery mechanisms, and phishing protection. 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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