SMART CONTRACT AUDIT REPORT

for

xRaise

Prepared By: Xiaomi Huang

PeckShield

December 15, 2023

Document Properties

Client	xRaise
Title	Smart Contract Audit Report
Target	xRaise
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	December 15, 2023	Xuxian Jiang	Final Release
1.0-rc1	December 9, 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 smart contract source code of the `xRaise` 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 audited protocol 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 xRaise

`xRaise` allows users to create an account abstraction wallet secured with device passkeys (`webauthn`). Users can create wallet using e-main or social login, connect different devices to one smart contract, add friends who can help to recover wallet by approving recovery request. It allows to pay fees in stable tokens, create permissioned sessions for games, and batch several transactions in one. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of xRaise

Item	Description
Name	xRaise
Website	https://xraise.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 15, 2023

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

- https://bitbucket.org/raise-finance/xraise-wallet-contracts.git (ddab458)

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

- https://bitbucket.org/raise-finance/xraise-wallet-contracts.git (31cb04c)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 `xRaise` 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 logic, 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	2	■ ■
Low	4	■ ■ ■ ■
Informational	0	
Total	6	

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 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key xRaise Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incomplete getUserWalletInfo() Logic in AccountFactoryDiamond	Coding Practices	Resolved
PVE-002	Low	Duplicate Session Avoidance in SessionFacet	Business Logic	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-004	Medium	Revisited withdrawSubscriptions() Logic in SubscriptionFacet	Business Logic	Resolved
PVE-005	Low	Revisited Modifiers in SocialRecoveryFacet	Business Logic	Resolved
PVE-006	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incomplete getUserWalletInfo() Logic in AccountFactoryDiamond

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AccountFactoryDiamond`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `xRaise` protocol has a key `AccountFactoryDiamond` contract that is the factory of all user account abstraction wallets. This contract defines a public function to retrieve the saved user authentication info on wallet sign in. Our analysis on this function indicates it exports incomplete wallet information.

To elaborate, we show below the code snippet of the related routine, i.e., `getUserWalletInfo()`. This routine is expected to return the set of devices registered for login. However, the returned `devices` information is empty. To fix, we need to add the following statement at the end of the function body, `devices = AccountFacet(payable(walletAddress)).devices();`.

```
    function getUserWalletInfo(bytes32 emailHash, uint32 deviceId)
        public view
        returns (
            address walletAddress, bytes32 credentialId, bytes32 credentialIdSlot2,
                bytes32 credentialIdCarry, bool inRecovery, uint32[] memory devices
        )
    {
        walletAddress = walletAddresses[emailHash];

        if (walletAddress != address(0)){
            LoginInfo memory plaformLogin;
            (plaformLogin) = AccountFacet(payable(walletAddress)).platformLogins(
                deviceId);
            credentialId = plaformLogin.credentialId;
            credentialIdSlot2 = plaformLogin.credentialIdSlot2;
            credentialIdCarry = plaformLogin.credentialIdCarry;

            (inRecovery,) = SocialRecoveryFacet(walletAddress).recoveryInfo();
        }
    }
```

Listing 3.1: `AccountFactoryDiamond::getUserWalletInfo()`

Recommendation Improve the above routine to fully retrieve the expected information.

Status This issue has been fixed in the following commit: a431c87.

3.2 Duplicate Session Avoidance in SessionFacet

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SessionFacet`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `xRaise` protocol has a `SessionFacet` contract to support the creation of multiple access sessions. While examining the session-related management, we notice it does not detect and block possible session duplicates.

To elaborate, we show below the related routines: `createSession()` and `deleteSession()`. As the names indicate, the first routine is used to create a new session and the second one is designed to delete an existing session. We notice that the session creation does not prevent a duplicate session from being created in the `sessionPubKeys` array (line 54). A duplicate session causes unnecessary confusion for the session management.

```
37      function createSession(
38          address sessionPubKey,
39          address[] calldata addressesToCallList,
40          uint256 totalGasLimit,
41          SessionPeriod period
42      )
43          external
44      {
45          require(msg.sender == address(this), "Only self user allowed");
46
47          SessionInfo storage session = sessionStorage().sessions[sessionPubKey];
48
49          session.sessionPubKey = sessionPubKey;
```

```
        session.addressesToCallList = addressesToCallList;
        session.gasLeft = totalGasLimit;
        session.expiredAt = sessionPeriodToSeconds(period);
        session.deleted = false;
        sessionStorage().sessionPubKeys.push(sessionPubKey);

        for (uint256 i = 0; i < addressesToCallList.length; i++) {
            sessionStorage().addressesToCall[sessionPubKey][addressesToCallList[i]] =
                true;
        }
    }

    function deleteSession(address signer) external {
        require(msg.sender == address(this), "Only self user allowed");

        sessionStorage().sessions[signer].deleted = true;
        for (uint256 i = 0; i < sessionStorage().sessionPubKeys.length; i++) {
            if (sessionStorage().sessionPubKeys[i] == signer) {
                sessionStorage().sessionPubKeys[i] = sessionStorage().sessionPubKeys[
                    sessionStorage().sessionPubKeys.length - 1];
                sessionStorage().sessionPubKeys.pop();
            }
        }
    }
```

Listing 3.2: `SessionFacet::createSession()/deleteSession()`

Recommendation Revise the above routines to ensure the same session may not be duplicated.

Status This issue has been fixed in the following commit: 9d48c69.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- 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 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.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 `withdrawToken()` routine in the `SubsidizingPaymaster` contract. If the `USDT` token is supported as the payment `token`, the unsafe version of `IERC20(token).transfer(msg.sender, amount)` (line 49) 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)!

```
48    function withdrawToken(address token, uint256 amount) public onlyOwner {
49        IERC20(token).transfer(msg.sender, amount);
50    }
```

Listing 3.4: `SubsidizingPaymaster::withdrawToken()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note this issue also affects other related functions in `TokenPaymaster`, `SubsidizingPaymaster`, and `WalletAirdrop`.

Status This issue has been fixed in the following commit: 2927e59.

3.4 Revisited withdrawSubscriptions() Logic in SubscriptionFacet

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SubscriptionFacet`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

`xRaise` has a `SubscriptionFacet` feature that allows to add/update/remove user subscriptions. While reviewing the related subscription-removing logic, we notice the implementation should be revisited.

To elaborate, we show below the implementation of the related `withdrawSubscriptions()` routine. This routine is defined public and can be called by any one to withdraw active subscriptions. For each active subscription, it may only called once in one subscription period. However, the subscription token should be sent to the `subscription.to`, not `msg.sender`.

```
    function withdrawSubscriptions() public {
        for (uint256 i = 0; i < subscriptionStorage().subscriptionIds.length; i++) {
            SubscriptionInfo storage subscription = subscriptionStorage().subscriptions[
                subscriptionStorage().subscriptionIds[i]];
            if (subscription.isActive && !subscription.isDeleted && block.timestamp >=
                subscription.nextChargeTime) {
                IERC20(subscription.token).transfer(msg.sender, subscription.price);
                subscription.nextChargeTime = block.timestamp + subscriptionPeriodToTime
                    (subscription.period);
            }
        }
    }
```

Listing 3.5: `SubscriptionFacet::withdrawSubscriptions()`

Recommendation Revise the above logic to properly handle subscription withdrawal.

Status This issue has been fixed in the following commit: 31cb04c.

3.5 Revisited Modifiers in SocialRecoveryFacet

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SocialRecoveryFacet
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the xRaise protocol has a unique feature in support social recovery when a user wallet device is lost. To faciliate the logic and code management, the contract defines a number of modifiers. Our analysis on these modifiers shows they can be improved.

To elaborate, we show below the related modifiers: onlyUserGuardian, onlyExternalGuardian, and onlyGuardian. As the names indicate, the first modifier is used to validate the caller is the user guardian, the second modifier checks the caller is the external guardian, and the last one verifies the wallet is in the recovery mode. It comes to our attention the first two modifiers share the same code while the last modifier always yields false. To fix, we need to remove the third modifier and re-defines the second one.

```
    modifier onlyUserGuardian {
        require(socialRecoveryStorage().isUserGuardian[keccak256(abi.encodePacked(msg.
            sender))], "only user guardian");
        _;
    }

    modifier onlyExternalGuardian {
        require(socialRecoveryStorage().isUserGuardian[keccak256(abi.encodePacked(msg.
            sender))], "only external guardian");
        _;
    }

    modifier onlyGuardian {
        require(!socialRecoveryStorage().inRecovery, "wallet is in recovery");
        require(socialRecoveryStorage().inRecovery, "wallet is not in recovery");
        _;
    }
```

Listing 3.6: SocialRecoveryFacet::onlyUserGuardian/onlyExternalGuardian/onlyGuardian

Recommendation Revise the above modifiers to ensure they are intended per design.

Status This issue has been fixed in the following commit: 2ed99fd.

3.6 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `xRaise` protocol, there is a privileged account (`owner`). This account plays critical roles in governing and regulating the protocol-wide operations (e.g., configure protocol parameters and upgrade protocol implementations). Our analysis shows that the privileged account needs to be scrutinized. In the following, we use the `AccountFactoryDiamond` contract as an example and show the representative functions potentially affected by the privileged account.

```
    function manageAllowedGuardians(address[] memory guardians, bool isAllowed) public
        onlyOwner {
        for(uint256 i = 0; i < guardians.length; i++){
            allowedGuardians[guardians[i]] = isAllowed;
            emit GuardianStatusSet(guardians[i], isAllowed);
        }
    }

    /// @dev functions for debug purposes to update notifier
    function setNotifier(address newNotifier) public onlyOwner {
        notifier = Notifier(newNotifier);
    }
```

Listing 3.7: Example Privileged Operations in `AccountFactoryDiamond`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the contract 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 addition, the user wallet makes use of the `Diamond` proxy to faciliate the management of various functionalities. The proxy may be upgraded, which is guarded with the owner as well.

Recommendation Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirmed that all the privileged accounts will be multi-sig wallets.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `xRaise` protocol, which allows users to create an account abstraction wallet secured with device passkeys (`webauthn`). Users can create wallet using e-main or social login, connect different devices to one smart contract, add friends who can help to recover wallet by approving recovery request. It allows to pay fees in stable tokens, create permissioned sessions for games, and batch several transactions in 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 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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