SMART CONTRACT AUDIT REPORT

for

MixToEarn

Prepared By: Xiaomi Huang

PeckShield

August 12, 2023

Document Properties

Client	MixToEarn
Title	Smart Contract Audit Report
Target	MixToEarn
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc	August 12, 2023	Xuxian Jiang	Final Release
1.0-rc	August 11, 2023	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 smart contract source code of the MixToEarn protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About MixToEarn

MixToEarn employs an advanced privacy mixer that utilizes zkSNARKs, a cryptographic protocol enabling privacy without compromising transaction verifiability. Users deposit their cryptocurrency into the privacy mixer, breaking the link between sender and receiver addresses using zkSNARKs and Merkle trees. This ensures transactional privacy through compact proofs of element inclusion. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of MixToEarn

Item	Description
Name	MixToEarn
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 12, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit only covers the Mixer1 contract.

- https://github.com/MixToEarn/smart-contracts.git (6ca51a4)

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

- https://github.com/MixToEarn/smart-contracts.git (6ca51a4)

1.2 About PeckShield

PeckShield Inc. [8] 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 the OWASP Risk Rating Methodology [7]:

- 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 checklist of 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	DeltaPrimeLabs 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

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) [6], 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 Logic	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 `MixToEarn` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	1	■
Low	2	■ ■
Informational	0	
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 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 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key MixToEarn Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited deposit() Logic in Mixer1	Business Logic	Resolved
PVE-002	Low	Improved Validation on Protocol Parameters	Coding Practices	Resolved
PVE-003	Low	Suggested Immutable/Constant Use in Mixer1	Coding Practices	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited deposit() Logic in Mixer1

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Mixer1`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Mixer1` contract allows users to deposit supported assets and the deposit will be charged for `1%` fee. Our analysis shows the current deposit logic needs to be revised.

To elaborate, we show below the related `deposit()` routine. Notice that it allows the deposit more than the specified `LIMIT` and the user is limited to withdraw `LIMIT - ParticipationFee`. Moreover, it comes to our attention that the deposit logic makes a low-level call to the contract itself, i.e., `payable(address(this)).call()` (line 61) and this low-level call is always reverted. The reason is that the contact does not implement the `receive()` or `fallback()` handlers to accept the native coin. With that, we suggest to remove this specific low-level call and refund the user if the user sends extra tokens (`msg.value-LIMIT`).

```
45     function deposit(uint256 identityCommitment) public payable {
46 
47         if(LIMIT > msg.value){
48             revert("Insufficient inventory");
49         }
50         if(CommitmentState[identityCommitment] == true){
51             revert("It is used Commitment");
52         }
53 
54 
55          //calculate fee
56         uint ParticipationFee = LIMIT * FEE / 10000;
57         uint Total = LIMIT - ParticipationFee;
```

```
            (bool success,) = payable(address(this)).call{value : Total}("");
             (bool succesd,) = payable(LAYAER).call{value : ParticipationFee}("");

            verifiyer.addMember(IndexId, identityCommitment);

        CommitmentState[identityCommitment]=true;
        CommitmentList.push(identityCommitment);
        WithAble[identityCommitment] = Total;

        emit Deposit(identityCommitment, block.timestamp);
    }
```

Listing 3.1: `Mixer1::deposit()`

Recommendation Revise the above routine to remove the low-level call as well as refund extra funds that have been sent in.

Status The issue has been fixed by this commit: `6ca51a4`.

3.2 Improved Validation on Protocol Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Mixer1`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `MixToEarn` protocol is no exception. Specifically, if we examine the `Mixer1` contract, it has defined a number of protocol-wide risk parameters, such as `FEE` and `LAYAER`. In the following, we show an example routine that allows for the `FEE` change.

```
    function setPutFee(uint128 _newFee) public onlyOwner() returns(bool){
        FEE = _newFee;
        return true;
    }
```

Listing 3.2: Mixer1::setPutFee()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `FEE` may charge unreasonably high fee in the user deposits, hence incurring cost to borrowers or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `6ca51a4`.

3.3 Suggested Immutable/Constant Use in Mixer1

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Mixer1`
- Category: Coding Practices [4]
- CWE subcategory: CWE-561 [2]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `MixToEarn` protocol, we observe there are several variables that need not to be updated dynamically. Examples include `verifiyer`, `IndexId`, `LIMIT`, and `FEE`. They can be declared as `immutable` or `constant` for gas efficiency.

```
contract Mixer1 {

    IVerifier public verifiyer;

    uint256 public IndexId;
    uint LIMIT = 1000000000000000000;//1
    uint[] CommitmentList;
    uint128 FEE = 100;//1%
```

```
    address LAYAER ;
    address OWNER;

    bool withdraw_able = false;
    ...
}
```

Listing 3.3: Example States Defined in `Mixer1`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status The issue has been fixed by this commit: `6ca51a4`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `MixToEarn` protocol, which employs an advanced privacy mixer that utilizes `zkSNARKs`, a cryptographic protocol enabling privacy without compromising transaction verifiability. Users deposit their cryptocurrency into the privacy mixer, breaking the link between sender and receiver addresses using `zkSNARKs` and `Merkle` trees. This ensures transactional privacy through compact proofs of element inclusion. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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

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

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