Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.07.01, the SlowMist security team received the SoulWallet team's security audit application for email-approver, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.

Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit

Serial Number	Audit Class	Audit Subclass
		External Call Function Security Audit
		Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-
17	Circuit Trusted Setup Risks	-
18	Overflow of Circuit Operations	-
19	Input Signal Cracking	-
20	Input Signal Leakage	-

3 Project Overview

3.1 Project Introduction

EIP-1271 style approver by zk verifying email

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Cross-chain replay attack risks	Replay Vulnerability	Information	Acknowledged
N2	Missing zero-address check in constructor	Others	Information	Acknowledged
N3	Boolean constants can be used directly	Others	Suggestion	Acknowledged
N4	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N5	Groth16 trusted setup risks	Circuit Trusted Setup Risks	Suggestion	Acknowledged

4 Code Overview

4.1 Contracts Description

Circom circuits:

https://github.com/SoulWallet/email-approver/blob/main/packages/circuits

Solidity smart contracts:

https://github.com/SoulWallet/email-approver/tree/main/packages/contracts/src

commit: 848ac10106560fd1f5a92edb21d80cf6c38c3295

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

DKIMRegistry			
Function Name	Visibility	Mutability	Modifiers

DKIMRegistry			
<Constructor>	Public	Can Modify State	Ownable
isDKIMPublicKeyHashValid	Public	-	-
setDKIMDomainName	Public	Can Modify State	onlyOwner
scheduleSetDKIMPublicKeyHash	Public	Can Modify State	onlyOwner
executeSetDKIMPublicKeyHash	Public	Can Modify State	onlyOwner
cancelSetDKIMPublicKeyHash	Public	Can Modify State	onlyOwner
scheduleSetDKIMPublicKeyHashes	Public	Can Modify State	onlyOwner
revokeDKIMPublicKeyHash	Public	Can Modify State	onlyOwner

EmailApprover			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
getEmailApproverInfo	External	-	-
_isValidProof	Internal	-	-
approve	Public	Can Modify State	-
isValidSignature	External	-	-
_authorizeUpgrade	Internal	Can Modify State	-

EmailApproverFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
proxyCode	External	-	-

EmailApproverFactory			
_proxyCode	Private	-	-
createEmailApprover	External	Can Modify State	-
getEmailApproverAddress	Public	-	-
_calcSalt	Private	-	-

4.3 Vulnerability Summary

[N1] [Information] Cross-chain replay attack risks

Category: Replay Vulnerability

Content

The `_isValidProof` function in the EmailApprover.sol contract verifies the validity of a transaction based on the provided parameters and the state of the DKIMRegistry. However, the verification logic does not check the transaction's chain or network. This means:

1.A user can generate a valid proof on a testnet.

2.The same proof can be replayed on the mainnet, as the signals and proof remain unchanged across different chains.

- packages/contracts/src/EmailApprover.sol

```
   function _isValidProof(uint256[8] memory proof, bytes32 pubkeyHash, bytes32
senderDomainHash, bytes32 approvedHash)
        internal
        view
        returns (bool)
   {
        // 1. Verify DKIM key
        // Note: this currently is not compitable with the current DKIMRegistry
        require(dkimRegistry.isDKIMPublicKeyHashValid(senderDomainHash, pubkeyHash),
"invalid dkim signature");


        uint256[6] memory signals;
        signals[0] = uint256(pubkeyHash);
        signals[1] = uint256(senderDomainHash);
```

```
        signals[2] = uint256(senderCommitment);
        signals[3] = uint256(uint160(address(this)));
        // split bytes32 hash into two parts
        signals[4] = uint256(approvedHash) >> 128;
        signals[5] = uint256(approvedHash) & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
        // Verify Dkim proof
        return verifier.verifyProof(
            [proof[0], proof[1]], [[proof[2], proof[3]], [proof[4], proof[5]]],
[proof[6], proof[7]], signals
        );
        //...
    }
```

Solution

Incorporate the chain ID as an additional parameter in the signing and verification process to ensure the proof is only valid on the chain it was generated.

Status

Acknowledged; The generation rules for `approveHash` are defined in the social recovery module and are replay-protected in the reference context.

[N2] [Information] Missing zero-address check in constructor

Category: Others

Content

In the constructor of the contract, there is a missing zero-address check for the `_approverImpl` parameter. This oversight can lead to critical functionality issues or potential vulnerabilities if the contract is initialized with an invalid zero address.

- packages/contracts/src/EmailApproverFactory.sol

```
    constructor(address _approverImpl) {
        _APPROVERIMPL = _approverImpl;
    }
```

Solution

Add a check in the constructor to ensure that `_approverImpl` is not the zero address. If the provided address is zero, the constructor should revert.

Status

Acknowledged

[N3] [Suggestion] Boolean constants can be used directly

Category: Others

Content

Boolean constants can be used directly and do not need to be compare to true or false.

- packages/contracts/src/DKIMRegistry.sol

```
    function scheduleSetDKIMPublicKeyHash(bytes32 domainNameHash, bytes32
publicKeyHash) public onlyOwner {
        require(dkimPublicKeyHashes[domainNameHash][publicKeyHash] == false, "already
registered");
    //...
    }
```

Solution

`require(!dkimPublicKeyHashes[domainNameHash][publicKeyHash], "already registered");`

Status

Acknowledged

[N4] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

In DKIMRegistry.sol contract, owner can:

```
setDKIMDomainName
scheduleSetDKIMPublicKeyHash
executeSetDKIMPublicKeyHash
cancelSetDKIMPublicKeyHash
scheduleSetDKIMPublicKeyHashes
revokeDKIMPublicKeyHash
```

These actions can affect or control the user's invocation of the zk account.

Solution

In the short term, using a multi-sig wallet to manage admin permissions can mitigate single point of failure well in the early stages of the protocol. But it does not alleviate the privilege escalation issue. In the long run, transferring admin ownership to community governance is a good practice, as it can greatly increase trust from community users in the protocol.

Status

Acknowledged; The project party plans to set the owner to a multi-signature address after deploying the contract.

[N5] [Suggestion] Groth16 trusted setup risks

Category: Circuit Trusted Setup Risks

Content

This project uses Groth16 protocol to verify private input values, but there are some security issues with Groth16 protocol. The Groth16 protocol produces a toxic waste during trust setup that can become a backdoor to the system and threaten system security if it is not safely discarded.

Solution

Securely generate trust parameters using, for example, MPC.

Status

Acknowledged; The project will use PSE for trust setup.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002407040002	SlowMist Security Team	2024.07.01 - 2024.07.04	Medium Risk

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 2 medium risk, 1 low risk, 2 suggestion vulnerabilities.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist