P2P.ORG ETH ALLOCATION SHARE SECURITY AUDIT REPORT

Nov 04, 2024

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals
- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal
Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal
Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals
- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

This project implements a Merkle tree-based contract for securely airdropping ETH to eligible recipients. The contract, called `MerkleAirdrop`, uses a Merkle tree to efficiently verify and distribute airdrop amounts to multiple recipients. It allows the contract owner to set a Merkle root, which represents the list of eligible addresses and their corresponding ETH amounts.

1.4 Project Dashboard

Project Summary

Title	Description
Client	P2P.org
Project name	ETH Allocation Share
Timeline	21.10.2024 - 22.10.2024
Number of Auditors	3

Project Log

Date	Commit Hash	Note
21.10.2024	dd05a846fc427b92b9fd60c8aeed990b16270286	Commit for the audit
22.10.2024	f6bc15e5d658024dd8537ece49655f86c3cb9111	Commit for the re-audit

Project Scope

The audit covered the following files:

File name	Link
src/MerkleAirdrop.sol	MerkleAirdrop.sol

Deployments

File name	Contract deployed on mainnet	Comment

File name	Contract deployed on mainnet	Comment
MerkleAirdrop.sol	0xA2431B...eDD50F6e	

1.5 Summary of findings

Severity	# of Findings
Critical	0
High	0
Medium	0
Low	3

ID	Name	Severity	Status
L-1	`setMerkleRoot()` Input Parameter Check	Low	Fixed
L-2	Missing Zero Address Check	Low	Fixed
L-3	Inability to Claim Additional Tokens After `merkleRoot` Update	Low	Acknowledged

1.6 Conclusion

During the audit, we thoroughly tested critical attack vectors and verified the following:

1. **Potential Reentrancy Attack on the `claim` Function**: The contract follows the Checks-Effects-Interactions pattern, effectively preventing reentrancy attacks. In the `claim` function, the state variable `claimed[msg.sender]` is updated before the ETH is transferred to the claimant, ensuring that no reentrancy can occur because the user's claim status is marked before any external interaction.

2. **Correctness of the Merkle Proof or Leaf Calculation**: The contract calculates the Merkle tree leaf correctly using `keccak256(abi.encodePacked(msg.sender, _amount))`. This ensures that the leaf corresponds to the recipient and the claimed amount. The Merkle proof verification is implemented properly using OZ `MerkleProof.verify`, preventing manipulation of the proof or any miscalculation of the leaf.

3. **Front-running User Claims**: The contract's design prevents front-running attacks since the Merkle tree leaf is computed using `msg.sender`, binding the claim directly to the caller's address. This ensures that no malicious actor can front-run a legitimate claim by submitting a transaction with a higher gas price, as only the intended recipient's address will pass the verification process.

4. **Appropriate Access Controls for Critical Functions**: The `setMerkleRoot` and `recoverEther` functions are secured with the `onlyOwner` modifier, ensuring that only the contract owner can update the Merkle root or recover unclaimed Ether.

5. **Handling of Unclaimed Ether**: Any unclaimed Ether in the contract will not be lost because the contract includes a `recoverEther()` function. This function allows the owner to withdraw the remaining balance of the contract at any time.

2.FINDINGS REPORT

2.1 Critical

Not Found

2.2 High

Not Found

2.3 Medium

Not Found

2.4 Low

L-1	`setMerkleRoot()` Input Parameter Check
Severity	Low
Status	Fixed in f6bc15e5

Description

`MerkleAirdrop.setMerkleRoot()` MerkleAirdrop.sol#L26-L29 updating the `merkleRoot` to the same value it already holds, which could make it more difficult to detect potential errors in the root updates.

The issue is classified as **Low** severity because it does not introduce direct security risks but may impact the protection against owner's errors.

Recommendation

We recommend adding a check to ensure that the new `merkleRoot` is different from the current value before updating it.

Client's Commentary

> Fixed in f6bc15e5

L-2	Missing Zero Address Check
Severity	Low
Status	Fixed in f6bc15e5

Description

This issue has been identified within the `recoverEther` function of the `MerkleAirdrop` contract. The function MerkleAirdrop.sol#L49-L52 whether the `_recipient` address is a zero address before attempting to transfer the contract's balance. This could result in the loss of funds if a zero address is mistakenly passed as the recipient.

The issue is classified as **Low** severity because in rare cases it can lead to funds loss.

Recommendation

We recommend adding a check to ensure that `_recipient` is not a zero address before transferring the balance.

Client's Commentary

> Fixed in f6bc15e5

L-3	Inability to Claim Additional Tokens After `merkleRoot` Update
Severity	Low
Status	Acknowledged

Description

If a user has already claimed their airdrop before the `merkleRoot` is updated, and in the new version of the tree the user is entitled to more tokens, MerkleAirdrop.sol#L32 the added amount under the new root.

The issue is classified as **Low** severity because it impacts only those users who have claimed before the `merkleRoot` update, and this scenario is not highly likely to occur frequently.

Recommendation

We recommend storing how many tokens a user has already claimed and allowing an additional claim if the new version of the tree entitles the user to more tokens.

Client's Commentary

> Acknowledged. We don't plan to update the `merkleRoot` once the claiming has started.
> It would make the accounting messy anyway since the new root might imply lower amounts than already claimed.
> In case of emergency, we plan to use `recoverEther` and deploy a new instance of the `MerkleAirdrop` contract instead.

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes