SALUS SECURITY

OCT 2024

CODE SECURITY ASSESSMENT

CAI

Overview

Project Summary

- Name: CAI
- Platform: EVM-compatible chains
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	CAI
Version	v1
Type	Solidity
Dates	Oct 08 2024
Logs	Oct 08 2024

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	1
Total Low-Severity issues	0
Total informational issues	2
Total	3

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 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 with 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.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Centralization risk	Medium	Centralization	Pending
2	Lack of input parameter check in setMerkleRoot	Informational	Data Validation	Pending
3	Inaccurate error message	Informational	Code Quality	Pending

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Centralization risk

Severity: Medium	Category: Centralization
Target: - CAI.sol - MerkleClaim.sol	

Description

In the `CAI` and `MerkleClaim` contract, there exists a privileged role called `owner`. The `owner` has the authority to execute some key functions such as `pause`, `unpause`, `mint` and `updateCaiContract`.

If the `owner`'s private key is compromised, an attacker could trigger these functions to mint lots of CAI tokens.

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

2.3 Informational Findings

2. Lack of input parameter check in setMerkleRoot

Severity: Informational	Category: Data Validation
Target: - MerkleClaim.sol	

Description

In MerkleClaim contract, the owner will build one Merkle tree and users can claim their rewards via this Merkle tree.

The Merkle tree leaves are generated using the `user` and `amount`. It's possible to generate two identical Merkle trees with the same Merkle root in different rounds. If the same Merkle root is used again, users will be unable to claim their rewards.

Merkle.sol:L45-L74

```
function setMerkleRoot(
    bytes32 _merkleRoot,
    uint256 _claimPeriodEnd,
    uint256 _allocation,
    bool _isEther
) external payable onlyOwner {
    ...
    claims[_merkleRoot] = Claim({
        merkleRoot: _merkleRoot,
        claimPeriodEnd: _claimPeriodEnd,
        allocation: _allocation,
        balance: _allocation,
        isEther: _isEther
    });
    ...
}
```

Merkle.sol:L76-L91

```
function claim(
    bytes32[] calldata merkleProof,
    uint256 amount,
    bytes32 _merkleRoot
) external nonReentrant {
    ...
    if (claimed[_merkleRoot][user]) revert AlreadyClaimed();
    bytes32 leaf = keccak256(abi.encodePacked(user, amount));
    ...
}
```

Recommendation

Add input parameter validation for function `setMerkleRoot`.

3. Inaccurate error message

Severity: Informational	Category: Code Quality

Target:
- MerkleClaim.sol

Description

The error message in the code snippet is incorrect. When `msg.value` exceeds `_allocation`, the error message should not describe the issue as "Insufficient."

MerkleClaim.sol:L45-L74

```
function setMerkleRoot(
    bytes32 _merkleRoot,
    uint256 _claimPeriodEnd,
    uint256 _allocation,
    bool _isEther
) external payable onlyOwner {
    ...
    if (_isEther) {
        require(msg.value == _allocation, "Insufficient Ether sent");
    } else {
        require(msg.value == 0, "Ether not needed for ERC20 claims");
        caiContract.transferFrom(msg.sender, address(this), _allocation);
    }
    ...
}
```

Recommendation

Replace `Insufficient Ether sent` with `Wrong Ether sent`.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
CAI.sol	3438a7376ae76dc0fff442d20cc7089667c21bcb
MerkleClaim.sol	9aeafac32f0e2f77dd88c1f033cbde4af8b896b2