PoorPleb Audit

October 2022

By CoinFabrik

v202209

Executive Summary

CoinFabrik was asked to audit the contracts for the PoorPleb project. No git repository was provided with the source code.

The scope for this audit includes and is limited to the following file:

- `PoorPleb.sol`: It contains the `PoorPleb` contract. It is a standard ERC20 contract with the extra option to claim tokens for some addresses. Its sha256, calculated using the sha256sum utility, is `f3cfb4d232154c82008dec2710f9f553f503abbf1416ba1c233d2a7f860ed638.`

No other files were audited. Its dependencies are assumed to work according to their documentation. Also, no tests were reviewed for this audit.

During this audit we found no critical issues, one medium issue and several minor issues. Also, two enhancements were proposed.

Fixes were published in the https://github.com/PoorPleb/PoorPlebSM.git git repository, commit `a81549c9ef980813178af0fd0115b247a9e80071`. All issues were resolved by the development team. All enhancements were implemented by the development team.

Methodology

CoinFabrik was provided with the source code. Our auditors spent one week auditing the source code provided, which includes understanding the context of use, analyzing the boundaries of the expected behavior of each contract and function, understanding the implementation by the development team (including dependencies beyond the scope to be audited) and identifying possible situations in which the code allows the caller to reach a state that exposes some vulnerability. Without being limited to them, the audit process included the following analyses:

- Arithmetic errors
- Outdated version of Solidity compiler
- Race conditions
- Reentrancy attacks
- Misuse of block timestamps
- Denial of service attacks
- Excessive gas usage
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Insufficient validation of the input parameters

- Incorrect handling of cryptographic signatures
- Centralization and upgradeability

After delivering a report with our findings, the development team had the opportunity to comment on every finding and fix the issues they considered convenient. Once fixed and/or commented, our team ran a second review process to verify that the changes to the code effectively solve the issues found and do not unintentionally add new ones. This report includes the final status after the second review.

Findings

In the following table we summarize the security issues we found in this audit. The severity classification criteria and the status meaning are explained below. This table does not include the enhancements we suggest to implement, which are described in a specific section after the security issues.

ID	Title	Severity	Status
ME-01	Multiple claimPA() Calls	Medium	Resolved
MI-01	No Zero Check	Minor	Resolved
MI-02	Phases Overlap	Minor	Resolved
MI-03	Solidity Version Pinning	Minor	Resolved

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of, but might be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.

Issues Status

An issue detected by this audit has one of the following statuses:

- **Unresolved**: The issue has not been resolved.
- **Acknowledged**: The issue remains in the code, but is a result of an intentional decision.
- **Resolved**: Adjusted program implementation to eliminate the risk.
- **Partially resolved**: Adjusted program implementation to eliminate part of the risk. The other part remains in the code, but is a result of an intentional decision.
- **Mitigated**: Implemented actions to minimize the impact or likelihood of the risk.

Critical Severity Issues

No issues found.

Medium Severity Issues

ME-01 Multiple claimPA() Calls

Location:

- `PoorPleb.sol: 60-62,66`

In the second phase, the `claimPA()` may be called several times after the owner calls the `setPA()` function, minting the PA reward several times to different addresses.

Recommendation

Add an internal boolean variable `claimedPA` to check if the PA has claimed its reward, set it to `true` when the `claimPA()` function is executed and change the `require` statement in line 66 to use it. This change would also allow an executor of the `claim()` function to run the `claimPA()` function.

Status

Resolved. Recommendation followed.

Minor Severity Issues

MI-01 No Zero Check

Location:

- `PoorPleb.sol: 60-62,66`

The `setPA()` function does not check if the new account to be set is zero. This may lead to an invalid PA.

Recommendation

Check that `account != 0` before assigning it.

Status

Resolved. Recommendation followed.

MI-02 Phases Overlap

Location:

- `PoorPleb.sol: 40,65`

If a block happens to have the `allowClaimAllDate` timestamp, both `claim()` and `claimPA()` will be able to be called, allowing a call to `claim()` after a call to `claimPA()`.

Recommendation

Change the condition in the require statement in line 65 to `block.timestamp > allowClaimAllDate`. Note the lack of "=".

Status

Resolved. Both checks in `claim()` and `claimPA()` were changed to fail if `block.timestamp == allowClaimAllDate`. This is not an issue, given that `claimPA()` can be called in the next blocks.

MI-03 Solidity Version Pinning

Location:

- `PoorPleb.sol: 2`

The `pragma solidity` statement in line 2 allows the PoorPleb contract to be compiled with newer versions of the solidity compiler. This may introduce unintended bugs when new compilers are released.

Recommendation

Pin the solidity version in the pragma, not allowing newer compilers.

Status
Resolved. Recommendation followed.

Enhancements

These items do not represent a security risk. They are best practices that we suggest implementing.

ID	Title	Status
EN-01	Version Control	Implemented
EN-02	Automated Tests	Implemented

EN-01 Version Control

The audited code is not under version control, such as git or mercurial.

Recommendation
Track the versions of the developed code using a version control system.

Status
Implemented. The corrected version is published in https://github.com/PoorPleb/PoorPlebSM.git, commit `a81549c9ef980813178af0fd0115b247a9e80071`.

EN-02 Automated Tests

While the development team provided us with some automated tests for the `PoorPleb` contract, they do not run properly.

Recommendation
Fix the tests and check that they have the proper coverage.

Status
Implemented. Tests in the corrected version pass.

Other Considerations

The considerations stated in this section are not right or wrong. We do not suggest any action to fix them. But we consider that they may be of interest to other stakeholders of the project, including users of the audited contracts, token holders or project investors.

Centralization

The pleb address (`PA` in the source code) can be set by the contract owner multiple times. The development team informed us that they intend to renounce the ownership of the contract as soon as the contract is launched.

Upgrades

The audited contract cannot be upgraded.

Privileged Roles

These are the privileged roles that we identified on the `PoorPleb` contract.

Owner

The address with the owner role can set the pleb address (`PA`). Also, given that the contract is a standard OpenZeppelin `Ownable`, it can transfer the ownership to a different address or renounce its ownership, effectively making the contract ownerless. It must be noted that the development team informed us that they intend to run with no admin keys, so they will renounce the contract ownership as soon as the contract is launched.

PA

Any address can call the `claimPA()` function after 69 days. When this function is called, the address with the PA role will get `N * 420_369.0` tokens, where `N` is the number of addresses that successfully called the `claim()` function during the first phase.

Contract Phases

The `PoorPleb` contract has two different phases.

First Phase

The first phase happens for the first 69 days. During that phase, any address that can provide a proof to the `claim()` function will be awarded 1420369 tokens. Each address may do the claim once in this phase

Second Phase

After those 69 days:

- addresses can no longer claim their rewards using the `claim()` function.
- any address can call the `claimPA()` function, awarding the PA `N * 420_369.0` tokens to the `PA`. And addresses

Minting

PP tokens are minted in 3 different situations.

Construction

When the `PoorPleb` contract is deployed, 1_666_725_026_387.72 tokens are minted and assigned to the deployer address. Those 1_666_725_026_387.72 tokens correspond to over 1_100_000 claims.

Claim

During the first phase, when an address successfully calls the `claim()` function, 1_420_369.0 tokens are minted and awarded to the caller of the function.

PA Claim

During the second phase, on a successful call of the `claimPA()` function, N * 420_369.0 are minted and awarded to the pleb address. N is the number of addresses that successfully called the `claim()` function in the first phase.

Changelog

- 2022-10-11 – Initial report.
- 2022-10-17 – Check fixes in commit `a81549c9ef980813178af0fd0115b247a9e80071`.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the PoorPlebs project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.