QuillAudits

Audit Report
March, 2022

For

Artemis

Contents

Contents

Overview

Artemis

Scope of the Audit

The scope of this audit was to analyze the Artemis token smart contract's codebase for quality, security, and correctness.

Artemis Contract: https://github.com/thetrees1529/artemis-contracts

Branch: main

Commit: dab5a9461fa95cad30a6b10ec905bb7d21f1e33c

Fixed In: 5276180d3e043d0a132a814b315e3274c1b8ce27

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiplying
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	1	1	0
Closed	1	2	1	6

Functional Testing

Complete functional testing report has been attached below:

Compiler output:

```
Brownie v1.18.1 - Python development framework for Ethereum

  File "brownie/_cli/__main__.py", line 64, in main
    importlib.import_module(f"brownie._cli.{cmd}").main()
  File "brownie/_cli/compile.py", line 50, in main
    proj = project.load()
  File "brownie/project/main.py", line 768, in load
    return Project(name, project_path)
  File "brownie/project/main.py", line 188, in __init__
    self.load()
  File "brownie/project/main.py", line 200, in load
    self._sources = Sources(contract_sources, interface_sources)
  File "brownie/project/sources.py", line 58, in __init__
    raise NamespaceCollision(
NamespaceCollision: Multiple contracts or interfaces with the same name
  IDOWithJustCollateral: contracts/IDOWithJustCollateral.sol, contracts/IDOWithJustWhitelist.sol
```

The contract name should be updated to IDOWithJustWhitelist in contracts/IDOWithJustWhitelist.sol.

Some of the tests performed are mentioned below:

- should be able to whitelist the users
- should allow users to collateralize.
- should allow users to contribute as per the conditions (whitelisting, collateralization etc.)
- should allow users to take refunds after the buying period ends.
- should allow contribution only in the defined time limit else revert
- should allow users to claim and take refunds in case of overflows
- should revert if the non-owner address withdraws the contributions and unsold tokens.
- should revert if non-owner address tries to pause/unpause
- should revert if non owner address tries to forceWithdraw
- should revert if non owner tries to forceReturn

Issues Found – Code Review/Manual Testing

High severity issues

1. **Usage Of transfer Instead Of safeTransfer**

   The ERC20 standard token implementation functions return the transaction status as a Boolean. It's good practice to check for the return status of the function call to ensure that the transaction was successful. It is the developer's responsibility to enclose these function calls with require() to ensure that, when the intended ERC20 function call returns false, the caller transaction also fails. However, it is mostly missed by developers when they carry out checks. In effect, the transaction would always succeed, even if the token transfer didn't.

   Recommendation
   Use the safeTransfer function from the safeERC20 Implementation or put the transfer call inside an assert or require to verify that it returned true.

   Status: Fixed
   Auditor's Comment: SafeERC20 is used for all IERC20 related functions.

Medium severity issues

2. **Centralization Risk**

   [#L21] function forceWithdraw() and forceReturn() can be called by the contract owner to withdraw the funds contributed as well as the sale tokens. This poses a risk for the token holders to end up with no tokens.

```
104     function forceWithdraw(uint amount) external override onlyOwner {
105         (bool success,) = msg.sender.call{value: amount}("");
106         require(success);
107     }
108
109     function forceReturn(uint amount) external override onlyOwner {
110         _parameters.token.transfer(msg.sender, amount);
111     }
112
```

Recommendation
We advise the client to handle the governance account carefully to avoid any potential hack. We also advise the client to consider the following solutions: with reasonable latency for community awareness on privileged operations; Multisig with community-voted 3rd-party independent co-signers; DAO or Governance module increasing transparency and community involvement;

Status: Acknowledged
Auditor's Comment: Artemis has considered using Multisig wallet for these functions.

3. **Avoid using .transfer() to transfer Ether**

   Although transfer() and send() have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. For reference, read more.

   Recommendation
   Use .call{ value: ... }("") instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection.

   Status: Fixed
   Auditor's Comment: .transfer() is replaced by .call{ value: ... }("").

4. **Check-Effect-Interaction pattern not followed**

   Calling _collateralInfo.token.transferFrom() is a call to an external contract.If there are vulnerable external calls, reentrancy attacks could be conducted because these two functions have state updates and event emits after external calls. The scope of the audit would treat the third-party implementation as a black box and assume its functional correctness. However, third parties may be compromised in the real world that leads to assets lost or stolen.

```
16      function collateralise() external override {
17          require(block.timestamp < _parameters.buyingStartsAt, "Buying has not already started.");
18          require(!hasRole(COLLATERALISED_ROLE, msg.sender), "Already collateralised.");
19          _collateralInfo.token.transferFrom(msg.sender, address(this), _collateralInfo.amount);
20          _grantRole(COLLATERALISED_ROLE, msg.sender);
21          emit Collateralised(msg.sender);
22      }
23      function refundCollateral() external override {
24          require(block.timestamp >= _parameters.buyingEndsAt, "Buying has not ended yet.");
25          require(hasRole(COLLATERALISED_ROLE, msg.sender), "Not collateralised.");
26          _collateralInfo.token.transfer(msg.sender, _collateralInfo.amount);
27          _revokeRole(COLLATERALISED_ROLE, msg.sender);
28          emit CollateralRefunded(msg.sender);
29      }
```

Recommendation
As per solidity security recommendation, the functions should first update the contract states and then interact with external contracts.
Please refer solidity documentation here: https://docs.soliditylang.org/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern

Please find the recommended implementation below:

```
function collateralise() external override {
  require(block.timestamp < _parameters.buyingStartsAt, "Buying has not already started.");
  require(!hasRole(COLLATERALISED_ROLE, msg.sender), "Already collateralised.");

  _grantRole(COLLATERALISED_ROLE, msg.sender);
  _collateralInfo.token.transferFrom(msg.sender, address(this),   _collateralInfo.amount);

  emit Collateralised(msg.sender);
}

function refundCollateral() external override {
  require(block.timestamp >= _parameters.buyingEndsAt, "Buying has not ended yet.");
  require(hasRole(COLLATERALISED_ROLE, msg.sender), "Not collateralised.");

  _revokeRole(COLLATERALISED_ROLE, msg.sender);
  _collateralInfo.token.transfer(msg.sender, _collateralInfo.amount);

  emit CollateralRefunded(msg.sender);
}
```

Status: Fixed
Auditor's Comment: These functions use a reentrancy guard which protects the code against possible reentrancy.

Low severity issues

5. **Floating Pragma**

   The contract makes use of the floating-point pragma ^0.8.0. Contracts should be deployed using the same compiler version and flags that were used during the testing process. Locking the pragma helps ensure that contracts are not unintentionally deployed using another pragma, such as an obsolete version that may introduce issues in the contract system.

```
//SPDX-License-Identifier: Unlicensed
pragma solidity ^0.8.0;
```

   Recommendation
   Consider locking the pragma version. It is advised that floating pragma not be used in production.

   Status: Fixed
   Auditor's Comment: Pragma version is now Locked to 0.8.13

6. **Renounce Ownership**

   Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities on his behalf. The renounceOwnership function is used in smart contracts to renounce ownership. Otherwise, if the contract's ownership has not been transferred previously, it will never have an Owner, which is risky.

   Recommendation
   It is advised that the Owner cannot call renounceOwnership without first transferring ownership to a different address. Additionally, if a multi-signature wallet is utilized, executing the renounceOwnership method for two or more users should be confirmed. Alternatively, the Renounce Ownership functionality can be disabled by overriding it.

   Status: Acknowledged

Informational issues

7. Public functions that are never called by the contract should be declared external to save gas. Functions claimableOf() and refundableOf() are declared public in contracts/IDO.sol.

   Status: Fixed

8. **State Variable Default Visibility**

   Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

   IDOWithCollateral.sol
   [#L9] CollateralInfo _collateralInfo;

   IDO.sol
   [#L8] Parameters _parameters;
   [#L9] GlobalStats _globalStats;
   [#L10] mapping(address=>UserStats) _userStats;

   Recommendation
   Variables can be specified as being public, internal or private. Explicitly define visibility for all state variables.
   Ref: https://swcregistry.io/docs/SWC-108

   Status: Fixed

9. **Variables declared as uint instead of uint256**

   To favor explicitness, consider changing all instances of uint into uint256 in the entire codebase.

   Status: Fixed

10. **Incorrect Error Message**

    The error message in IDOWithCollateral.sol describe the error incorrectly:

```
require(block.timestamp < _parameters.buyingStartsAt, "Buying has not
already started.");
```

    We recommend changing it to “Buying has already started”.

    Status: Fixed

11. **Order of Functions and pragma directives**

    Ordering helps readers identify which functions they can call and find the constructor and fallback definitions easier.

    Recommendation
    Pragma should be declared at the top of the files after the license.
    Functions should be grouped according to their visibility and ordered:
    - constructor
    - fallback function (if exists)
    - external
    - public
    - internal
    - private

    Within a grouping, place the view and pure functions last. It is also advised to use solhint to format the smart contracts to improve readability.

    Status: Fixed

12. **Missing Docstrings**

It is extremely difficult to locate any contracts or functions, as they lack documentation. One consequence of this is that reviewers' understanding of the code's intention is impeded, which is significant because it is necessary to accurately determine both security and correctness. They are additionally more readable and easier to maintain when wrapped in docstrings. The functions should be documented so that users can understand the purpose or intention of each function, as well as the situations in which it may fail, who is allowed to call it, what values it returns, and what events it emits.

Recommendation

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if those are not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Status: Fixed

Closing Summary

Some issues of High, Medium and Low severity were found, which needs to be fixed by Developers. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of Artemis. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Artemis team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
March, 2022

For

Artemis

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com