SMARTHUB INNOVATIONS

origintrail

TRACE TOKEN AUDIT REPORT

DECEMBER 1st - 11th 2023

by

SmartHub Innovations
#SmartAudits

CONTENTS

EXECUTIVE SUMMARY

Project Name: Trace Token

Overview: Purpose-built Protocol for Supply Chains Based on Blockchain.

Timeline: 1st to 11th December, 2023

Method: Manual Review, Functional Testing, Automated Testing etc.

Scope of Audit: The scope of this audit was to analyze Trace Token codebase for quality, security, and correctness.

Codebase:
https://etherscan.io/token/0xaa7a9ca87d3694b5755f213b5d04094b8d0f0a6f#code

	High	Medium	Low	Informational
Open Issues	2	3	1	7
Acknowledged Issues	0	0	0	0
Resolved & Closed	0	0	0	0

TYPES OF SEVERITIES

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 fifixed before moving to a live environment.

Medium
The issues marked as medium severity usually arise because of errors and defificiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fifixed.

Low
Low-level severity issues can cause minor impact and or are just warnings that can remain unfifixed for now. It would be better to fifix 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.

TYPES OF ISSUES

Open
Security vulnerabilities identifified that must be resolved and are currently unresolved.

Resolved
These are the issues identifified in the initial audit and have been successfully fifixed.

Acknowledged
Vulnerabilities which have been acknowledged but are yet to be resolved.

CHECKED VULNERABILITIES

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

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 analyzed 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 security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerability or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of 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 possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Hardhat , Hardhat Team, Solhint, Mythril, Slither, Solidity statistic analysis.

ISSUES FOUND - MANUAL TESTING/CODE REVIEW

CRITICAL/HIGH SEVERITY ISSUES

1. Incorrect Constructor Name

Severity; Critical

```
/**
 * @dev The Ownable constructor sets the original `owner`
 * account.
 */
ftrace | funcSig
function Ownable() public { owner = msg.sender; }
```

Description:

The constructor in the `Ownable` contract uses the contract name instead of the constructor keyword. This discrepancy can lead to unexpected behavior and is a critical issue. In Solidity versions equal to or greater than 0.4.22, using the contract name as the constructor is deprecated and can cause problems.

Remediation

Update the constructor in the Ownable contract to use the constructor keyword instead of the contract name. Replace function `Ownable() public { ... }` with `constructor() public { ... }.`
This ensures compatibility with newer Solidity versions and avoids potential issues during deployment.

2. No Emit Declaration in transferOwnership Function

Severity: Critical

Description:

In the `Ownable` contract, the `transferOwnership` function allows the current owner to transfer control of the contract to a new owner. However, there is no emit statement to trigger the `OwnershipTransferred` event, which can lead to a lack of transparency and monitoring for ownership changes. Events are crucial in Ethereum smart contracts to log and notify external systems about important state changes.

Remediation

To address this issue, it is essential to add the `emit` statement after the `OwnershipTransferred` event is triggered. Here is the recommended modification to the `transferOwnership` function.

```
function transferOwnership(address newOwner) onlyOwner public {
    require(newOwner != address(0));
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
}
```

By adding `emit OwnershipTransferred(owner, newOwner);` after the `OwnershipTransferred` event, the smart contract will emit this event whenever ownership is transferred, providing a clear and auditable record of such changes on the blockchain.

MEDIUM SEVERITY ISSUES

1. Gas Limitations:

Severity: Medium

```
ftrace | funcSig | CodiumAI: Options | Test this function
function allocateRestOfTokens() onlyOwner public{
  require(totalSupply > TOTAL_NUM_TOKENS.div(2));
  require(totalSupply < TOTAL_NUM_TOKENS);
  require(!mintingFinished);
  mint(wallet, bountyReward);
  mint(advisorsAndPreICO,  preicoAndAdvisors.div(5));
  mint(wallet, liquidityPool);
  mint(wallet, futureDevelopment);
  mint(this, teamAndFounders.sub(CORRECTION));
  mint(this, preicoAndAdvisors.mul(4).div(5));
}
```

Description:

The `allocateRestOfTokens` function in the `TracToken` contract may pose gas limitations due to the substantial number of tokens being minted within a single transaction. Ethereum imposes a gas limit on transactions, and if the gas consumption exceeds this limit, the transaction will fail. Minting a large number of tokens in one transaction can potentially surpass the gas limit.

Potential Risks:
Transaction Failure: If the gas consumption surpasses the network's limit, the transaction will be reverted, leading to a failure in the minting process.

Increased Gas Costs: Large transactions often result in higher gas costs, and users may need to pay more transaction fees to ensure miners process their transactions.

Remediation

Consider breaking down the token allocation process into smaller transactions to mitigate potential gas limitations. This approach allows for the gradual minting of tokens, reducing the risk of exceeding gas limits and enhancing the chances of successful execution.

2.Gas Usage in Loop

Severity: Medium

Description

Functions like `**withdrawTokenToFounders**` and `**withdrawTokensToAdvisors**` in the `**TracToken**` contract contain conditional statements based on time within loops. This structure may result in higher gas consumption, especially if the loop iterates over an extended period or performs complex operations.

Potential Risks:

High Gas Costs: Gas costs increase with the number of iterations in a loop and the complexity of operations performed within each iteration.

Possible Transaction Reversion: If the gas limit is exceeded, the entire transaction may be reverted, leading to undesirable consequences.

Remediation

Carefully assess the gas consumption of these functions, especially considering the loop structures. Consider optimizing the logic within the loops and testing gas usage under various conditions to ensure the functions are efficient and cost-effective.

3.Reentrancy Risk

Severity: Medium

Description:

Functions like `withdrawTokenToFounders` in the `TracToken` contract involve multiple token transfers within a loop. This design may introduce a reentrancy risk if state changes are not performed before token transfers. Reentrancy occurs when an external contract is called before the state changes are completed, potentially leading to unexpected behavior.
Malicious contracts could exploit the reentrancy vulnerability, leading to unintended consequences and potential loss of funds.

Remediation

Ensure that state changes are executed before any token transfers within the loop to minimize the risk of reentrancy attacks. Use best practices for secure contract design to prevent reentrancy vulnerabilities by using reentrancy guard.

LOW SEVERITY ISSUES

1. Redundant Assert in SafeMath Library

```
library SafeMath {
 function mul(uint256 a, uint256 b) internal constant returns (uint256) {
   uint256 c = a * b;
   assert(a == 0 || c / a == b);
   return c;
 }
```

Description:

The `assert(a == 0 || c / a == b);` statement in the `mul` function of the SafeMath library is redundant. This assertion checks whether `a` is zero, but the multiplication operation in the line above already guarantees that `a` is non-zero.
Redundant code may lead to decreased code readability without providing additional safety.

Remediation:

Remove the redundant `assert` statement to simplify the code and improve readability. The multiplication itself ensures that a is non-zero, making the additional check unnecessary.

INFORMATIONAL ISSUES

1. Floating Pragma and Older Versions of Compiler

N/A (Fallback function is not present in the code)

Description:

The pragma solidity statement in the contract specifies a floating version (^0.4.18), allowing the compiler to use any version greater than or equal to 0.4.18. Additionally, the code explicitly targets an older version of the compiler (0.4.18), which might not benefit from the latest compiler optimizations, bug fixes, and security enhancements introduced in more recent versions of Solidity.

Targeting an older version of the compiler (0.4.18) might result in missed optimizations, bug fixes, and security patches introduced in later Solidity versions.
It is advisable to use a more recent and well-audited version of the compiler to benefit from the latest improvements and to ensure compatibility with the broader Ethereum ecosystem.

Remediation:

Specify a fixed pragma version that is tested and known to be compatible with the code. For example, replace `pragma solidity ^0.4.18;` with `pragma solidity 0.4.18;` to fix the version.

Consider upgrading to a more recent and stable version of the Solidity compiler. This ensures that the contract leverages the latest features, optimizations, and security enhancements provided by the Solidity development team.

2. No Emit Declaration in transfer Function

```
function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);

    // SafeMath.sub will throw if there is not enough balance.
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    // Missing emit declaration for the Transfer event
    Transfer(msg.sender, _to, _value);
    return true;
}
```

Description:

In the `BasicToken` contract, specifically in the `transfer` function, there is no explicit `emit` declaration for the `Transfer` event after the transfer has been completed. Emitting events is crucial in Ethereum contracts for transparency and allows external parties to track state changes.

Remediation:

Add the `emit` keyword before the `Transfer` event to explicitly emit the event after the transfer is completed. This ensures that the event is logged on the Ethereum blockchain, providing a clear record of token transfers.

The emit keyword is optional in Solidity versions starting from `v0.4.21`. However, for better code clarity and to adhere to modern Solidity practices, explicitly using `emit` is recommended when emitting events. This change does not affect the functionality but enhances the readability of the code and aligns with the current best practices.

3. No Emit Declaration in mint Function

Severity: Informational

Description:

In the `mint` function of the `MintableToken` contract, the `Mint` and `Transfer` events are not explicitly emitted after minting tokens. Events play a crucial role in providing transparency and allowing external systems to react to state changes. The absence of event emissions can result in a lack of visibility into token minting activities, hindering proper monitoring and auditability of the contract.

Remediation:

To address this issue, you should include the necessary event emissions in the mint function to provide a clear and transparent record of token minting activities. Add the following lines after the corresponding state changes:

```
emit Mint(_to, _amount);
emit Transfer(0x0, _to, _amount);
```

These lines emit the `Mint` and `Transfer` events, indicating the minting of tokens from the zero address (`0x0`) to the specified recipient address (`_to`). Including these events enhances the visibility and traceability of minting operations, facilitating better monitoring and analysis of the contract's behavior.

4. Magic Random Numbers

Severity: Informational

Description:

The code contains magic numbers, such as `720 days` , `630 days` , etc., without clear explanations or comments. Magic numbers can make the code less readable and maintainable since their significance may not be immediately apparent to someone reviewing the code.

Recommendation:

It is advisable to replace these magic numbers with named constants or provide comments explaining their significance. For example, instead of `720 days` , define a constant like `SECONDS_IN_720_DAYS` and use it in the code. This improves code readability and makes it easier for developers to understand the purpose of these time-related values.

```
uint256 constant SECONDS_IN_720_DAYS = 720 days;

// ...

if (now > startTime + SECONDS_IN_720_DAYS && founderAmounts[7] > 0) {
    // ...
}
```

By introducing named constants, the code becomes more self-explanatory, and future developers can easily comprehend the time-related conditions.

5. Inconsistency in Function Naming

Description:

There is inconsistency in the naming convention used for functions, specifically the naming of functions related to minting and finishing minting.

In the `TracToken` contract, there is a function named `endMinting`, which indicates the conclusion of the minting process.
In contrast, the function in the `MintableToken` contract that serves a similar purpose is named `finishMinting`. This inconsistency in naming may lead to confusion for developers and users alike, as they might expect consistent naming conventions for related functionalities.

Remediation:

To ensure clarity and maintainability, it is recommended to standardize the naming convention for functions related to minting and finishing minting. Choose a consistent naming pattern, such as using either `endMinting` or `finishMinting` across all relevant contracts. This consistency helps developers easily understand and navigate the codebase.

6. Wrong Code Layout

Severity: Informational

Description:

The code layout in the provided Solidity contract could be improved for better readability and maintainability. The organization of the code sections and functions appears to be suboptimal, potentially making it harder for developers to understand the structure of the contract.

Remediation:

Improve the code layout by following the solidity style guide https://docs.soliditylang.org/en/latest/style-guide.html

7. Redundant Function Override

Severity: Informational

Description:

In the `TracToken` contract, there are redundant overrides of the `transfer` and `transferFrom` functions. These functions are already defined in the parent contracts (`BasicToken` and `StandardToken), and the overrides in the TracToken contract don't introduce any additional logic or modifications.

Explanation:

Solidity allows contracts to override functions inherited from parent contracts. However, if the overriding function does not introduce any changes to the behavior of the parent function, it is considered redundant and can be omitted. Redundant overrides can clutter the code and may lead to confusion, especially for developers reviewing or maintaining the code.

Remediation:

To improve code clarity and reduce redundancy, consider removing the overridden `transfer` and `transferFrom` functions from the `TracToken` contract if they don't serve any specific purpose. If there is a need to extend or modify the behavior of these functions, then additional logic should be included in the overridden functions. Otherwise, removing the redundant overrides will make the code cleaner and easier to understand.

AUTOMATED TEST

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Trace Token codebase. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and Informational severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

SmartAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Trace Token Platform. 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 Trace Token Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

ABOUT SMARTAUDITS

SmartAudits is a secure smart contracts audit platform designed by SmartHub Innovations Technologies.
We are a team of dedicated blockchain security experts and smart contract auditors
determined to ensure that Smart Contract-based Web3 projects can avail the latest and best
security solutions to operate in a trustworthy and risk-free ecosystem.