TOKEN DELTA AUDIT REPORT

November, 2023

by

SmartHub Innovations
#SmartAudits

CONTENTS

EXECUTIVE SUMMARY

Project Name: Token Delta

Overview: Legally Binding Smart Contracts, Powered by AI.

Timeline: 25th November to 30th, November

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

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

Codebase:
https://etherscan.io/token/0x07e3c70653548b04f0a75970c1f81b4cbbfb606f#code

	High	Medium	Low	Informational
Open Issues	1	2	2	4
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. Constructor Function Safety

```
function HumanStandardToken(
    uint256 _initialAmount,
    string _tokenName,
    uint8 _decimalUnits,
    string _tokenSymbol
) {
```

Description:

The `HumanStandardToken` constructor lacks the `public` and `payable` modifiers, which can pose a security risk. Constructors should not be payable, and explicitly specifying the visibility is a good practice for clarity and security.

Constructors in Solidity are executed only once during contract deployment. Making a constructor payable could allow anyone to send Ether to the contract during deployment, potentially causing unintended behavior. Explicitly specifying the visibility of the constructor is recommended for clarity and to prevent any unexpected changes in future Solidity versions.

Remediation

Change function `HumanStandardToken(` to `constructor(` and ensure it is non-payable

MEDIUM SEVERITY ISSUES

1. Lack of Visibility for Functions (Lines 14-26):

```
function approveAndCall(address _spender, uint256 _value, bytes
_extraData) returns (bool success) {
```

```
require(_spender.call(bytes4(bytes32(sha3("receiveApproval(
address,uint256,address,bytes)"))), msg.sender, _value, this,
_extraData));
```

Description:

The `approveAndCall` function allows arbitrary contract calls, posing a potential security risk. Allowing external contracts to be called with arbitrary data can lead to unexpected behavior or exploits if the called contract executes malicious operations.

Remediation

Check and Restrict Calls: Implement a mechanism to check and restrict the allowed contracts or functions that can be called using `approveAndCall`. This could involve maintaining a whitelist of trusted contracts or using an access control mechanism.

Latest Solidity Version: Consider using the latest version of Solidity to leverage security improvements and best practices. Solidity evolves, and newer versions often come with enhancements in security and language features.

Use of Interface or Abstract Contracts: If possible, interact with external contracts through well-defined interfaces or abstract contracts with known functions. This can help ensure that the expected behavior is maintained.

2.Lack of Input Validation in approveAndCall

```
require(_spender.call(bytes4(bytes32(sha3("receiveApproval(
address,uint256,address,bytes)"))), msg.sender, _value, this,
_extraData));
```

Description

The `approveAndCall` function lacks proper input validation, exposing the contract to potential vulnerabilities arising from incorrect or malicious data provided in the `_extraData` parameter.

Detailed Analysis:
The `_spender` address is used in a low-level call without validating if it is a valid contract address.
No checks are performed on `_value` to ensure it's within reasonable bounds.
The function doesn't verify the length or content of `_extraData`.

Remediation

Check that `_spender` is a valid and non-zero address.
Utilize the `isContract` function to ensure that `_spender` is a contract.
Ensure that _value is greater than zero.
Confirm that _extraData is not empty if its length is checked by the called contract.

LOW SEVERITY ISSUES

1. Use of Deprecated "constant"

```
function balanceOf(address _owner) constant returns (uint256 balance)
{
function allowance(address _owner, address _spender) constant returns
(uint256 remaining) {
```

Description:

The constant keyword is deprecated in Solidity and has been replaced with view for functions that do not modify the state. Using the deprecated keyword may cause issues in future compiler versions.

Remediation:

Replace constant with view in the relevant functions.

Detailed Analysis:

In Solidity versions 0.4.17 and later, the constant keyword was officially deprecated in favor of view.
The constant keyword was originally used to indicate that a function does not modify the state, making it a pure function. In later versions, view was introduced to replace constant.

2. Constructor Function Name

```
function HumanStandardToken(
    uint256 _initialAmount,
    string _tokenName,
    uint8 _decimalUnits,
    string _tokenSymbol
) {
```

Description:

While not a critical concern, the use of `HumanStandardToken` as a constructor function name is unconventional. Constructors should typically share the same name as the contract.

The constructor function of the `HumanStandardToken` contract has a name that differs from the contract's name, which is not in line with common conventions. Conventionally, the constructor function should share the same name as the contract to enhance clarity and readability. Naming consistency makes it easier for developers to identify the constructor when reviewing the code.

Remediation:

To address this issue, it is recommended to rename the constructor function to match the contract name. In this case, renaming `HumanStandardToken` to Delta aligns with best practices.

INFORMATIONAL ISSUES

1. Fallback Function Absence

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

Description:

The contract lacks a fallback function (` function() external payable`). A fallback function is a special function that is executed when a contract receives Ether without specifying a function or when a function call fails. It is crucial for handling unexpected Ether transfers and preventing Ether from getting trapped in the contract.

Remediation:

Add a fallback function to handle unexpected Ether transfers.

2. Gas Consumption in approveAndCall

```
require(_spender.call(bytes4(bytes32(sha3("receiveApproval(
address,uint256,address,bytes)"))), msg.sender, _value, this,
_extraData));
```

Description:

The `approveAndCall` function includes a call to an external contract using the low-level call function. Performing external calls in this manner can consume a significant amount of gas, and it introduces potential vulnerabilities. Gas consumption should be carefully managed to avoid out-of-gas issues.

Remediation:

To address the gas consumption issue, consider the following remediation steps:

Gas Limitation: Limit the amount of gas sent along with the external call. This can be achieved using the gas stipend provided with the call function.
Example:
"require(_spender.call.gas(200000)(bytes4(bytes32(sha3("receiveApproval(address,uint256,address,bytes)"))), msg.sender, _value, this, _extraData));"
Adjust the gas value based on the requirements of the external contract.

Gas-Efficient Patterns: Break down complex operations into smaller steps to reduce gas consumption. Implement gas-efficient patterns, such as asynchronous calls or using separate transactions for certain operations.

Check Return Value: Check the return value of the call function to handle potential failures gracefully.

3. Deprecated Event Call Syntax

```
Transfer(msg.sender, _to, _value);
Approval(msg.sender, _spender, _value);
```

```
event Transfer(address _from, address _to, uint256 _value);
event Approval(address _owner, address _spender, uint256 _value);
```

Description:

The event calls for Transfer and Approval are using the old syntax without the emit keyword. Prior to Solidity version 0.5.0, event emissions were done without the emit keyword. However, in the latest versions, it is recommended to use emit for clarity and consistency.

Remediation:

Update the event calls to the new syntax by adding the emit keyword.

```
event Transfer(address indexed _from, address indexed _to, uint256 _value);
event Approval(address indexed _owner, address indexed _spender, uint256 _value);
```

```
emit Transfer(msg.sender, _to, _value);
emit Approval(msg.sender, _spender, _value);
```

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 Token Delta 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 Token Delta 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 Token Delta 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.