DeSoc

SMART CONTRACT AUDIT REPORT

Prepared by:

BlockAudit

Date Of Enrollment:

April 22nd, 2024 - April 30th, 2024

Visit : www.blockaudit.report

TABLE OF CONTENTS

SUMMARY

This Audit Report mainly focuses on the extensive security of **DESOC** Smart Contracts. With this report, we attempt to ensure the reliability and correctness of the smart contract by complete and rigorous assessment of the system's architecture and the smart contract codebase.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts producedby industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified inpublic;
- Provide more transparency on privileged activities once the protocol is live.

OVERVIEW

Project Summary

Project Name	DeSoc
Logo	
Platform	-
Language	Solidity
Contract Address	0xbb04Ac2227dd203F16c65fC1514c8E9C5C7c2dF0

File Summary

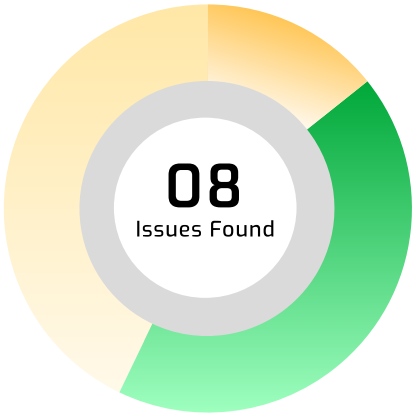
ID	File Name	Audit Status
DESOC	DeSoc.sol	Pass

Audit Summary

Date of Delivery	30 April 2024
Audit Methodology	Code Analysis. Automatic Assesment, Manual Review
Audit Result	Passed
Audit Team	BlockAudit Report Team

FINDINGS

Critical	0	0.0%
High	0	0.0%
Medium	1	12.75%
Low	3	37.5%
Informational	0	0.0%
Ownership	0	0.0%
Gas Optimization	4	50%



Vulnerability Findings Summary

ID	Type	Instances	Severity	Status
M-01	An Unbounded Loop On The Array Can Lead To DoS	-	Medium	Resolved
L-01	Avoid Using Floating Pragma	-	Low	Resolved
L-02	Lack of 2-step transfer of ownership.	-	Low	Acknowledged
L-03	Redundant hardhat/ console.sol Import	-	Low	Resolved
G-01	Pre-increment is cheaper than post-increment	-	Gas Optimisation	Resolved
G-02	Caching the array length outside a loop	-	Gas Optimisation	Resolved
G-03	Use Custom Errors instead of Revert Strings to save Gas	-	Gas Optimisation	Resolved
G-04	Explicitly initializing variables with their default values wastes gas	-	Gas Optimisation	Resolved

M-01

Type	An Unbounded Loop On The Array Can Lead To DoS
Severity	Medium
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

In the setTGEPassed() function of contract, a loop iterates through the list of recipient addresses, minting the respective amounts. If the list is very large, the transaction's gas cost could exceed the block gas limit and make it impossible to call this function at all.

Remediation

Add a limit for the length of the allocation recipients.

Snapshot

```
function setTGEPassed() external onlyOwner {
    require(TGETimestamp == 0, "TGE is already passed");
    TGETimestamp = block.timestamp;
    for (uint256 i = 0; i < allocations.length;) {
        _mint(allocations[i].recipient, allocations[i].amount);
        unchecked {
            i++;
        }
    }
    emit TGEPassed();
}
```

L-01

Type	Avoid Using Floating Pragma
Severity	Low
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

The Token contract uses floating pragma. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Consider replacing ^0.8.20 by 0.8.20

Snapshot

```
pragma solidity ^0.8.20;
```

L-02

Type	Lack Of 2-Step Transfer Of Ownership.
Severity	Low
File	DeSoc.sol
Line	-
Status	**Acknowledged**

Description

Ownable2Step is safer than Ownable for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer is only completed when the new owner accepts ownership.
Also, If the nominated EOA account is not valid, the owner may accidentally transfer ownership to an uncontrolled account, breaking all functions with the onlyOwner() modifier.

Remediation

Recommend considering implementing a two step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of ownership to fully succeed.

Snapshot

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

L-03

Type	Redundant Hardhat/Console.Sol Import
Severity	Low
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

hardhat/console is imported into the contracts. It is a redundant import and in the production environment, It is recommended to delete hardhat/console.

Remediation

Remove the import statement for hardhat/console.

Snapshot

```
import "hardhat/console.sol";
```

G-01

Type	Pre-Increment Is Cheaper Than Post-Increment
Severity	Gas Optimisation
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

Pre-increment(++i) costs less gas compared to post-increment (i++) for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration)

Remediation

Use ++i instead of i++ to increment the value of an uint variable

Snapshot

```
26:         unchecked {
27:             i++;
28:         }

38:         unchecked {
39:             i++;
40:         }
```

G-02

Type	Caching The Array Length Outside A Loop
Severity	Gas Optimisation
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

The setTGEPassed function contains loops that iterate over arrays. In each loop, the length of the array is read. However, the length of the array does not change during the loop, so it can be cached outside the loop. This would save 3 gas per iteration.

Remediation

The following code snippet shows how to cache the array length outside a loop:

```
uint256 length = tokens.length;
for (uint256 i = 0; i < length; ++i) {
  // do something with the token at index i
}
```

Snapshot

```
        for (uint256 i = 0; i < allocations.length;) {
            _mint(allocations[i].recipient, allocations[i].amount);
            unchecked {
                i++;
            }
```

G-03

Type	Use Custom Errors Instead Of Revert Strings To Save Gas
Severity	Gas Optimisation
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met). Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries).

Remediation

We suggest replacing revert strings with custom errors.

Snapshot

```
function setTGEPassed() external onlyOwner {
    require(TGETimestamp == 0, "TGE is already passed");
```

G-04

Type	Explicitly Initializing Variables With Their Default Values Wastes Gas
Severity	Gas Optimisation
File	DeSoc.sol
Line	-
Status	**Resolved**

Description

The default value of a uint256 variable is 0, so explicitly initializing i with 0 is unnecessary. This wastes gas, as the Solidity compiler has to store the value 0 in memory even though it is not used.

Remediation

Declare variables without initializing them. We can use uint i; instead of uint i = 0;

Snapshot

```
for (uint256 i = 0; i < allocations.length;) {
```

APPENDIX

Auditing Approach and Methodologies applied

The Block Audit Report team has performed rigorous testing of the project including the analysis of the code design patterns where we reviewed the smart contract architecture to ensure it is structured along with the safe use of standard inherited contracts and libraries. Our team also conducted a formal line by line inspection of the Smart Contract i.e., a manual review, to find potential issues including but not limited to

- Race conditions
- Zero race conditions approval attacks
- Re-entrancy
- Transaction-ordering dependence
- Timestamp dependence
- Check-effects-interaction pattern (optimistic accounting)
- Decentralized denial-of-service attacks
- Secure ether transfer pattern
- Guard check pattern
- Fail-safe mode
- Gas-limits and infinite loops
- Call Stack depth

In the Unit testing Phase, we coded/conducted custom unit tests written against each function in the contract to verify the claimed functionality from our client. In Automated Testing, we tested the Smart Contract with our standard set of multifunctional tools to identify vulnerabilities and security flaws. The code was tested in collaboration of our multiple team members and this included but not limited to;

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analyzing the complexity of the code in depth and in detail line-by-line manual review of the code.
- Deploying the code on testnet using multiple clients to run live tests.
- Analyzing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analyzing the security of the on-chain data.

Issue Categories:

Every issue in this report was assigned a severity level from the following:

Critical Severity Issues

Issues of Critical Severity leaves smart contracts vulnerable to major exploits and can lead to asset loss and data loss. These can have significant impact on the functionality/performance of the smart contract.
We recommend these issues must be fixed before proceeding to MainNet..

High Severity Issues

Issues of High Severity are not as easy to exploit but they might endanger the execution of the smart contract and potentially create crucial problems.
Fixing these issues is highly recommended before proceeding to MainNet.

Medium Severity Issues

Issues on this level are not a major cause of vulnerability to the smart contract, they cannot lead to data-manipulations or asset loss but may affect funtionality.
It is important to fix these issues before proceeding to MainNet.

Low Severity Issues

Issues at this level are very low in their impact on the overall functionality and execution of the smart contract. These are mostly code-level violations or improper formatting.
These issues can be remain unfixed or can be fixed at a later date if the code is redeployed or forked.

Informational Findings

These are finding that our team comes accross when manually reviewing a smart contract which are important to know for the owners as well as users of a contract.
These issues must be acknowledged by the owners before we publish our report.

Ownership Priviledges

Owner of a smart contract can include certain rights and priviledges while deploying a smart contract that might be hidden deep inside the codebase and may make the project vulnerable to rug-pulls or other types of scams.
We at BlockAudit believe in transparency and hence we showcase Ownership priviledges separately so the owner as well as the investors can get a better understanding about the project.

Gas Optimization

Solidity gas optimization is the process of lowering the cost of operating your Solidity smart code. The term "gas" refers to the level of processing power required to perform specific tasks on the Ethereum network.
Each Ethereum transaction costs a fee since it requires the use of computer resources. It will deduct a fee anytime any function in the smart contract is invoked by the contract's owner or users.

DISCLAIMER

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for the client to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that the client should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for the client to conduct the client's own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

By reading this report or any part of it, the client agrees to the terms of this disclaimer. If the client does not agree to the terms, then please immediately cease reading this report, and delete and destroy any/all copies of this report downloaded and/or printed by the client. This report is provided for information purposes only and stays on a non-reliance basis, and does not constitute investment advice. No one/ NONE shall have any rights to rely on the report or its contents, and BlockAudit and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives).

(BlockAudit) owes no duty of care towards the client or any other person, nor does BlockAudit claim any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and BlockAudit hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effects in relation to the report.

Except and only to the extent that it is prohibited by law, BlockAudit hereby excludes all liability and responsibility, and neither the client nor any other person shall have any claim against BlockAudit, for any amount or kind of loss or damage that may result to the client or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report.

The analysis of the security is purely based on the received smart contracts alone. No related/third-party smart contracts, applications or operations were reviewed for security. No product code has been reviewed.

Note: 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 **DeSoc** team put a bug bounty program in place to encourage further analysis of the smart contracts by other third parties

About BlockAudit

BlockAudit is an industry leading security organisation that helps web3 blockchain based projects with their security and correctness of their smart-contracts. With years of experience we have a dedicated team that is capable of performing audits in a wide variety of languages including HTML, PHP, JS, Node, React, Native, Solidity, Rust and other Web3 frameworks for DApps, DeFi, GameFi and Metaverse platforms.

With a mission to make web3 a safe and secure place BlockAudit is commited to provide it's partners with a budget and investor friendly security Audit Report that will increase the value of their projects significantly.

 *www.blockaudit.report*

 *team@blockaudit.report*

 *@BlockAudit*

 *github.com/Block-Audit-Report*