SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

For COMEARTH

Audited By
Satyabrata Dash & Aliasgar Kapadia

Report Prepared By
Aliasgar Kapadia

Table of Contents

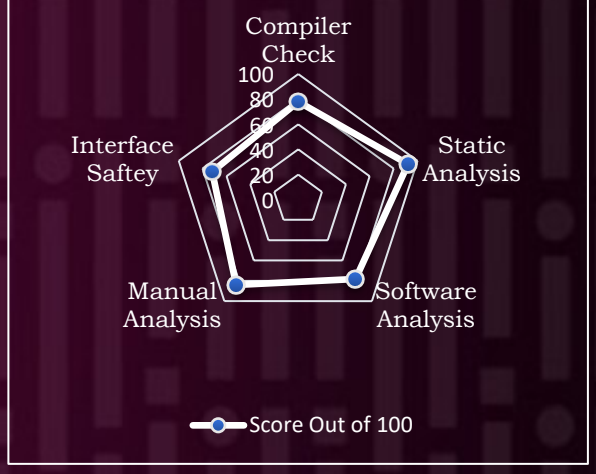
Disclaimer

The Secureverse team examined this smart contract in accordance with industry best practices. We made every effort to secure the code and provide this report. audits done by smart contract auditors and automated algorithms; however, it is crucial to remember that you should not rely entirely on this report. The smart contract may have flaws that allow for hacking. As a result, the audit cannot ensure the explicit security of the audited smart contracts. The Secureverse and its audit report do not encourage readers to consider them as providing any project-related financial or legal advice.

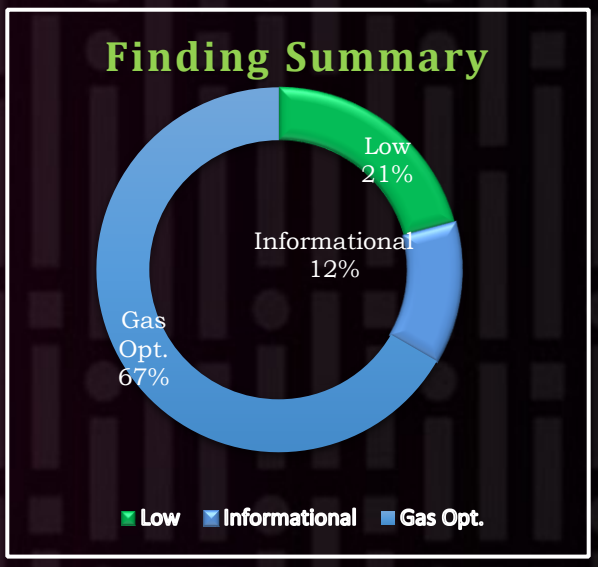
Executive Summary

Project Name	Comearth
Project Type	NFT & DeFi
Audit Scope	Check Security and code quality
Audit Method	Static and Manual
Audit Timeline	21st March, 2023 to 27th March 2023
Source Code	https://github.com/NFTically/comearth-smart-contracts/blob/dev/contracts

Security Score



Finding Summary



Issue Tracking Table

	High	Medium	Low	Informational	Gas Optimization
Open Issues	-	-	-	-	-
Acknowledged Issues	-	-	4	3	16
Resolved Issues	-	-	1	-	-

Types of Severities

- **High:** The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium:** The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
- **Low:** The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
- **Informational:** The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth.

Types of Issues

- **Open:** Security vulnerabilities identified that must be resolved and are currently unresolved.
- **Acknowledged:** The way in which it is being used in the project makes it unnecessary to address the vulnerabilities. This means that the way it has been acknowledged has no effect on its security.
- **Resolved:** These are the issues identified in the initial audit and have been successfully fixed.

Checked Vulnerabilities

- Re-entrancy
- Access control
- Denial of service
- Timestamp Dependence
- Integer overflow/Underflow
- Transaction Order Dependency
- Requirement Violation
- Functions Visibility Check
- Mathematical calculations
- Dangerous strict equalities
- Unchecked Return values
- Hard coded information
- Safe Ether Transfer
- Gas Consumption
- Incorrect Inheritance Order
- Centralization
- Unsafe external calls
- Business logic and specification
- Input validation
- Incorrect Modifier
- Missing events
- Assembly usage
- ERC777 hooks
- Token handling

Methods

Audit at Secureverse is performed by the experts and they make sure that audited project must comply with the industry security standards.

Secureverse audit methodology includes following key:

- In depth review of the white paper
- In depth analysis of project and code documentation.
- Checking the industry standards used in Code/Project.
- Checking and understanding Core Functionality of the Code.
- Comparing the code with documentation.
- Static analysis of the code.
- Manual analysis of the code.
- Gas Optimization and Function Testing.
- Verification of the overall audit.
- Report writing.

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

Static Analysis

Static analysis has been done by using the open source and state of the art automatic smart contract vulnerability scanning tools.

Manual Analysis

Manual analysis is done by our smart contract auditors' team by performing in depth analysis of the smart contract and identify potential vulnerabilities. Auditor also review and verify all the static analysis results to prevent the false positives identified by automated tools.

Gas Consumption and Function Testing

Function testing done by auditors by manually writing customized test cases for the smart contract to verify the intended behavior as per code and documentation. Gas Optimization done by reviews potential gas consumption by contract in production.

Tools and Platforms used for Audit

- Remix IDE
- Hardhat
- Mythril
- Truffle Team
- Solhint
- Solidityscan.com
- Slither
- Consensys Surya
- Open Zeppelin Code Analyzer
- Manticore

Findings

Low Severity Issues:

[L-1] Floating pragma used:

Reference:

File	Line Number
WhiteListManager.sol	3
TokenHolder.sol	3
TokenSale.sol	3

Description:
Consider to lock used Solidity version.

Status: **Resolved**

[L-2] There must be zero address check while setting important wallet addresses:

Reference:

File	Line Number
TokenHolder.sol	63-69
TokenSale.sol	97-107

Status: **Acknowledged**

[L-3] For transferring native token prefer to use `call()` instead of using `transfer()`. Transfer() now not recommended:

Reference:

File	Line Number
TokenSale.sol	184,189

Description:

.transfer() will relay 2300 gas and .call will relay all the gas.
If the receive/fallback function from the recipient proxy contract has complex logic, using .transfer will fail, causing integration issues.

Remediation:

Replace `.transfer` with `.call`. Note that the result of `.call` need to be checked.

Status: **Acknowledged**

[L-4] Ensure Both `monthCount` and `monthlyVestingPercentage[]`s length are equal:

Reference:

File	Line Number
TokenHolder.sol	132-146

```
function validateVestingSchedulePresetOrRevert(uint256 zeroDayVestingPercentage,
uint256 monthCount,
        uint256[] memory monthlyVestingPercentage) public view virtual {

        require(monthCount > 0, "TokenHolder: Vesting months should be >
0");  //@audit !=0

        // check percentage sums up to 100% i.e 100000 (PERCENTAGE_DIVISOR)
        uint256 vestingPercentageSum = zeroDayVestingPercentage;

        for (uint256 i = 0; i < monthCount; i++) {  //@audit ensure both monthCount
and arr len is same or not

            vestingPercentageSum += monthlyVestingPercentage[i];
        }

        require(vestingPercentageSum == PERCENTAGE_DIVISOR, "TokenHolder: Sum of
vesting percentages should be 100%");
    }
```

Status: **Acknowledged**

[L-5] Single point of failure:

Reference:

File	Line Number
TokenSale.sol	215-223

Description:

`setTokenSaleStatus()` is a onlyOwner function where token sale status changed, but point is, if it set to Stop option then can't resume.
As there always a chance of human error, so handle this with care. May be for Stopping option 2 step process could be helpful, so that owner can't by mistakenly call that function with incorrect parameter.

Status: **Acknowledged**

Informational Issues

[NC-1] Use scientific notation instead of long zeros number, helps in increasing readability:

Reference:

File	Line Number
TokenHolder.sol	25
TokenSale.sol	32, 34, 36, 521, 524

Remediation:
Instead of writing 100000, Write 10e5.

Status: **Acknowledged**

[NC-2] Absence of visibility for state variables

Reference:

File	Line Number
TokenHolder.sol	23, 25

Remediation:
Provide some visibility to state variable like public or private.

Status: **Acknowledged**

[NC-3] No need to write return()

Reference:

File	Line Number
WhiteListManager.sol	83

Description:
As function has `returns(var1, var2);`, No need to explicitly write ```return```

Status: **Acknowledged**

Gas Optimization

[G-1] Use of uints less than 32bytes incurs overhead:

Reference:

File	Line Number
WhiteListManager.sol	13, 15

Description:
When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

Status: **Acknowledged**

[G-2] Repeated require() statement could enclosed inside a modifier or public function and used further to save deployment gas:

Reference:

File	Line Number
WhiteListManager.sol	[113, 131, 182, 205, 231] and [135-136, 218]

Description:
The compiler will inline the function, which will avoid `JUMP` instructions usually associated with functions

Status: **Acknowledged**

[G-3] Instead of `public`, some functions could be `external`:

Reference:

File	Line Number
WhiteListManager.sol	174
TokenHolder.sol	81, 86, 94, 168, 234

Description:

Those Function which are not called inside contract can make as `external` instead of `public`.
Contracts are allowed to override their parents' functions and change the visibility from external to public and can save gas by doing so.

Status: **Acknowledged**

[G-4] Functions those will revert when called by normal user should marked as `payable`:

Reference:

File	Line Number
WhiteListManager.sol	55, 99, 110, 128, 180, 211, 226
TokenHolder.sol	86, 105, 115-116, 208
TokenSale.sol	215, 232, 248, 262, 276, 287, 299, 315, 329, 405, 422, 434, 481, 503, 510

Description:

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. The extra opcodes avoided are `CALLVALUE`(2), `DUP1`(3), `ISZERO`(3), `PUSH2`(3), `JUMPI`(10), `PUSH1`(3), `DUP1`(3), `REVERT`(0), `JUMPDEST`(1), `POP`(2), which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost.

Remediation:

Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Status: **Acknowledged**

[G-5] Assigning solidity variables with their default value costs extra gas:

Reference:

File	Line Number
WhiteListManager.sol	101, 68, 192, 233
TokenHolder.sol	140
TokenSale.sol	30, 357, 407

Status: **Acknowledged**

[G-6] Variables should be cached in memory and then further use it:

Reference:

File	Line Number
WhiteListManager.sol	70, 74, 76, 103
TokenHolder.sol	156

Description:

Like Cache array length outside of loop
If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Status: **Acknowledged**

[G-7] Operation `i++` costs more gas than `++i` same for subtraction as well:

Reference:

File	Line Number
WhiteListManager.sol	101, 68, 192, 233
TokenHolder.sol	140, 156
TokenSale.sol	407

Status: **Acknowledged**

[G-8] Arithmetic operations which will not overflow or underflow should mark `unchecked`:

Reference:

File	Line Number
WhiteListManager.sol	101, 68, 192, 233
TokenHolder.sol	120, 140, 142, 156, 158, 160, 216, 250
TokenSale.sol	407

Description:

Prior to Solidity 0.8.0, it by default comes with over/underflow checks on every arithmetic operation which costs more gas.

Remediation:

By making `uncheck` those arithmetic operation we can save gas.

Status: **Acknowledged**

[G-9] Zero address checks should preform through `assembly` will more cost effective:

Reference:

File	Line Number
WhiteListManager.sol	57
TokenHolder.sol	107, 210
TokenSale.sol	250, 264, 278, 301

Remediation:

Save 6 gas per instance by using assembly to check for address(0)

```
assembly { if iszero(_addr) { mstore(0x00, "zero address") revert(0x00, 0x20) }}
```

Status: **Acknowledged**

[G-10] Use `selfbalance()` instead of address(this).balance:

Reference:

File	Line Number
TokenSale.sol	505

Remediation:
You can use selfbalance() instead of address(this).balance when getting your contract's balance of ETH to save gas. Additionally, you can use balance(address) instead of address.balance() when getting an external contract's balance of ETH.

Saves 15 gas when checking internal balance, 6 for external

Status: **Acknowledged**

[G-11] Making CONSTANT variables `private` will save gas during deployment:

Reference:

File	Line Number
WhiteListManager.sol	11, 13, 15
TokenSale.sol	17

Description:
Saves 3406-3606 gas in deployment gas due to the compiler does not have to create non-payable getter functions for deployment calldata, does not have to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Status: **Acknowledged**

[G-12] Splitting `require()` statements that use &&, can save gas:

Reference:

File	Line Number
WhiteListManager.sol	113, 131, 135-136, 182, 205, 231, 218
TokenHolder.sol	214
TokenSale.sol	121, 483

Remediation:
Instead of using,

```
require(cond1 && cond2, error msg);
```

Try to use

```
require(cond1, error msg);
require(cond2, error msg);
```

Status: **Acknowledged**

[G-13] Instead of `> 0`, `!= 0` is more gas efficient:

Reference:

File	Line Number
WhiteListManager.sol	70, 133, 131, 182, 205, 231
TokenHolder.sol	135, 212, 214, 248
TokenSale.sol	119, 367, 377, 409, 436

Remediation:
Instead of using,

```
if( bal > 0){}
```

Try to use

```
if( bal != 0){}
```

Status: **Acknowledged**

[G-14] `structs` can be packed:

Reference:

File	Line Number
TokenHolder.sol	28-33

Remediation:
Some Variables could pack into a Single slot instead of two.

Status: **Acknowledged**

[G-15] Using bools for storage incurs overhead:

Reference:

File	Line Number
WhiteListManager.sol	101, 68, 192, 233
TokenHolder.sol	120, 140, 142, 156, 158, 160, 216, 250
TokenSale.sol	407

Description:

Booleans are more expensive than uint256 or any type that takes up a full word because each write operation emits an extra SLOAD to first read the slot's contents, replace the bits taken up by the Boolean, and then write back. This is the compiler's defense against contract upgrades and pointer aliasing, and it cannot be disabled.

Remediation:

Use uint256(1) and uint256(2) for true/false instead

Status: **Acknowledged**

[G-16] <x> += <y> Costs More Than <x> = <x> + <y> In Case Of State Variable:

Reference:

File	Line Number
TokenHolder.sol	250
TokenSale.sol	159, 162

Remediation:

use =+ or =- instead to save gas

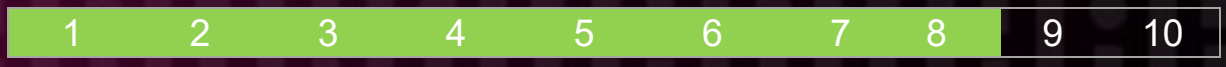
Status: **Acknowledged**

Automated Testing

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 audit, we examined the COMEARTH's smart contract with our framework, and we discovered several medium, low, and informational flaws in the smart contract. We have included solutions and recommendations in the audit report to improve the quality and security posture of the code. All of the findings and solutions have been acknowledged by the project team. In summary, we find that the codebase with the latest version greatly improved on the initial version. We believe that the overall level of security provided by the codebase in its current state is reasonable, so we have marked it as **Secure** and the Customer's smart contract has the following score: 8.5



About Secureverse

Secureverse is the Singapore and India based emerging Web3 Security solution provider. We at Secureverse provides the Smart Contract audit, Blockchain infrastructure Penetration testing and the Cryptocurrency forensic services with very affordable prices.

To Know More

Twitter: https://twitter.com/secureverse

LinkedIn: https://www.linkedin.com/company/secureverse/

Telegram: https://t.me/secureverse

Email Address: http://secureverse@protonmail.com/