SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

For

ORBIT HUMAN CARE (OHC)

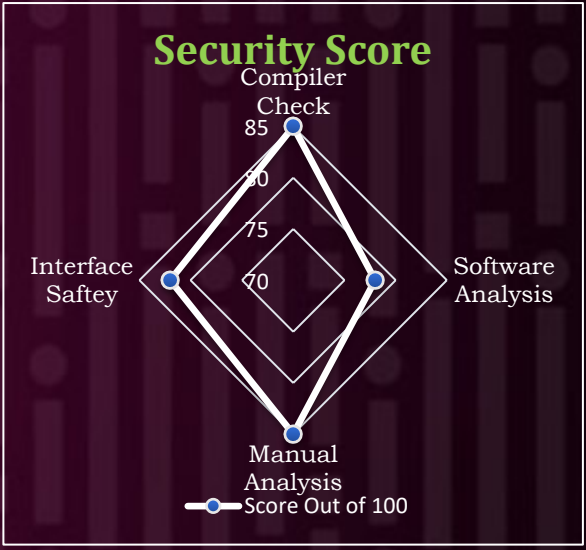
Table of Contents

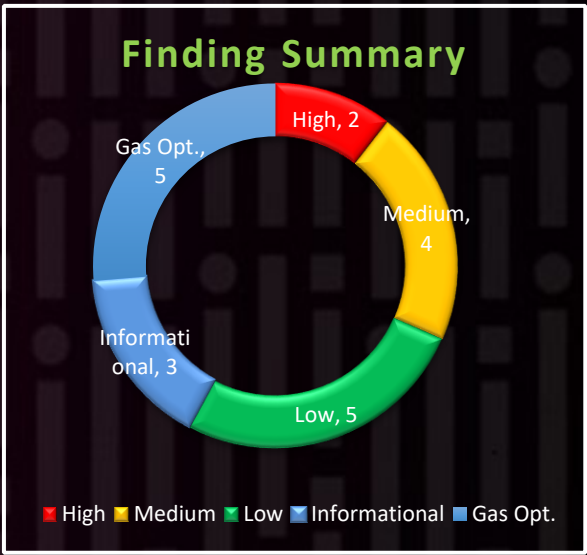
Disclaimer

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	Orbit Human Care (OCH)		
Project Type	RWA, Vesting		
Audit Scope	Check security and code quality		
Audit Method	Manual		
Code Version	V1	V1.1	V1.2
Audit Timeline	3-April-2024 to 13-April-2024	24-April-2024 to 26-April-2024	27-April-2024
Source Code	Vesting (VC)	Vesting (VC)	Vesting (VC)
	Vesting Proposal	Vesting Proposal	Vesting Proposal
	Vesting Marketing	Vesting Marketing	Vesting Marketing





Issue Tracking Table					
	High	Medium	Low	Informational	Gas Optimization
Open Issues	-	-	-	-	-
Acknowledged Issues	-	3	4	3	5
Resolved Issues	2	1	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.
- **Gas Optimization:** The issue also does not pose an immediate risk, but it is the process to making smart contracts more efficient, cost-effective, to enhance scalability and better user experience.

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
- Integer overflow/Underflow
- Transaction Order Dependency
- Requirement Violation
- Functions Visibility Check
- Mathematical calculations
- Dangerous strict equalities
- Unchecked Return values
- Hard coded information
- Malicious libraries
- Gas Consumption
- Incorrect Inheritance Order
- Centralization
- Unsafe external calls
- Business logic and specification
- Input validation
- Incorrect Modifier
- Missing events
- Assembly usage
- Improper or missing events
- 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.
- 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.

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.

Findings

High Severity Issues:

[H-01] Incorrect vesting calculation in `calculateReleaseToken()`

Reference:
1) Vesting(VC)#L57
2) Vesting_Marketing#L57

Description:
The `calculateReleaseToken()` within the `OCH_VESTING_MARKETING` contract is responsible for determining the amount of tokens eligible to vest when a user calls the `claim()` function. However, the function returns an incorrect amount of vested tokens.

```
function calculateReleaseToken() public view returns(uint256){
            uint256 returnAmount;

                if(OCH.balanceOf(address(this)) > 0){

                        uint256  time = block.timestamp - lastTimeClaimed;
                        uint256 perSecPercent
=((OCH.balanceOf(address(this))*percentRelase)/100)/(120); // 60*60*24*30
                        returnAmount += (time * perSecPercent);
                    }
                         return returnAmount;
    }
    function  claim() public  onlyOwner{
            require(block.timestamp >= lastTimeClaimed + 120 ," Claiming before 30
days"); // 60*60*24*30
            uint256  avaiableAmount = calculateReleaseToken();
            require(avaiableAmount <= OCH.balanceOf(address(this)) ,"insufficient
Contract Balanace");
                OCH.transfer(msg.sender,avaiableAmount);
                lastTimeClaimed = block.timestamp;

    }
```

This function computes the token release rate per second based on the contract's balance, obtained through the `balanceOf(address(this))`. However, the calculation method employed is flawed. Furthermore, it fails to adjust the release rate based on the remaining token balance after each claim, resulting in inaccurate token vesting calculations.

Proof of Concept:
This contract implements a vesting mechanism where tokens are gradually released over time. The intended behavior is that a certain percentage of tokens should be released every few minutes until the entire vesting period is complete.

For example, in this case, the vesting period is set to release 10% of the total token supply every 2 minutes, with the expectation that 100% of the tokens will be released after 20 minutes.

The issue arises in how the contract calculates the number of tokens to release. The contract mistakenly adjusts the release amount based on the remaining balance of tokens after each claim, rather than consistently releasing the intended percentage of tokens over time.

For example, let's say the initial token balance is 1,000,000 tokens. According to the vesting schedule, 10% of this balance (100,000 tokens) should be released every 2 minutes. However, due to the incorrect calculation, the contract calculates the release amount based on the remaining balance after each claim.

After the first claim, 100,000 tokens are released correctly. However, the remaining balance is now 900,000 tokens. Instead of continuing to release 10% of the initial balance (100,000 tokens) every 2 minutes, the contract incorrectly calculates 10% of the remaining balance (90,000 tokens) for the next release.

This results in a decreasing release rate over time, as the contract continues to base its calculations on the diminishing balance after each claim. As a result, the contract fails to release the full 100% of the tokens within the expected vesting period of 20 minutes.

Recommendation:
Calculate the `perSecPercent` value once and then utilize it consistently for all subsequent vesting periods thereafter. Can be calculated in `SetStartingPoint`().

Status: **Resolved**
Vesting(VC) resolved in V1.2
Intended for Vesting_Marketing

[H-01-v1.1] Token can stuck forever

Reference:
1) Vesting(VC)V1.1#L69-L124

Description:
The [H-01] issue has been **partially** resolved in the v1.1 code but now this leads to another vulnerability. `calculateReleaseToken()` is now **`calculateReturn()`** with updated code.

Now, if a user forgets to claim their vested tokens for any particular vesting round, the tokens allocated for that round will remain stuck in the contract indefinitely. Which is impossible to recover, even by the contract owner.

In `calculateReturn()` vesting schedule is set to occur every quarter, starting six. A fixed months after the fund allotment percentage of tokens is vested in each round, hardcoded into the contract. If a user fails to claim their tokens for a specific vesting round within the allocated time frame, the tokens allocated for that round become permanently stuck in the contract.

For instance, let's say user has claimed their token for first 3 round (i.e., 1 year), but misses claiming their 5% vested tokens during the 4th vesting round, which corresponds to a year after the fund allotment. Without a way to deal with tokens that haven't been claimed, they stay stuck in the contract forever. Nobody, even the owner of the contract, can get these tokens back. This means that the tokens are lost forever, even though they belong to the users. If users forget to claim their tokens, they end up being lost in the contract, and there's no way to get them back.

Recommendation:
Two possible solutions:

1) Implement a mechanism that enables the contract owner to recover unclaimed tokens after a specified period. Such a mechanism would allow the contract owner to retrieve the unclaimed tokens and redistribute them accordingly, ensuring that no tokens remain permanently stuck within the contract.

2) Code shared on GitHub. In this code new vesting strategy implemented by which user can claim and receive all the rewards that they are eligible. Tokens will not remain in contract anymore. Code Link

Status: **Resolved**

[H-02] Different time intervals can cause tokens to get stuck permanently

Reference:

1) Vesting(VC)#L51-L69

2) Vesting_Maketing# L51-L69

Description:

The *calculateReleaseToken()* uses a time interval of 120 seconds (2 minutes) to calculate *perSecPercent*. However, the *claim()* contains an assertion that checks if the current timestamp is at least 30 days (25,920,000 seconds) after the last time claimed. This difference in time intervals results in the *claim()* always reverting, as it's unlikely for the current timestamp to be 30 days after the last time claimed within a 2-minute interval.

```
function calculateReleaseToken() public view returns(uint256){
            uint256 returnAmount;

                if(OCH.balanceOf(address(this)) > 0){

                        uint256  time = block.timestamp - lastTimeClaimed;
                        uint256 perSecPercent
=((OCH.balanceOf(address(this))*percentRelase)/100)/(120); // 60*60*24*30
                        returnAmount += (time * perSecPercent);
                    }
                         return returnAmount;
    }
    function  claim(address user) public  onlyOwner{
            require(block.timestamp >= lastTimeClaimed + 60*60*24*30 ," Claiming
before 30 days"); // 60*60*24*30
            uint256  avaiableAmount = calculateReleaseToken();
            require(avaiableAmount <= OCH.balanceOf(address(this)) ,"insufficient
Contract Balanace");
                OCH.transfer(user,avaiableAmount);
                lastTimeClaimed = block.timestamp;

    }
```

Recommendation:

Adjust the time interval used in the *claim()* to match the 2-minute interval used in the *calculateReleaseToken()*. Or modify the calculateRelease*Token()* to use a time interval consistent with the 30-day requirement in the *claim()*.

Status: **Resolved in V1.2**

Medium Severity Issues:

[M-01] No checks on multisigner duplicates

Reference: Vesting_Proposal#L102-L106

Description:
OCH_VESTING_PROPOSAL contract lacks validation checks to prevent the addition of duplicate signers in the multisigner list. This allows the owner to add the same signer multiple times.

Recommendation:
Implement a check in the *addSigner()* to ensure that the signer address being added does not already exist in the multisigner list.

```
function addSigner(address signer) public onlyOwner {
    require(signer != address(0), "Invalid User Address");
    require(!isSigner(signer), "Signer already exists"); // Add this validation
    require(multisigner.length <= 10, "Limit Reached!! Cannot assign more signers");
    multisigner.push(signer);
}

function _isSigner(address signer) internal view returns (bool) {
    for (uint256 i; i < multisigner.length; ++i) {
        if (multisigner[i] == signer) {
            return true;
        }
    }
    return false;
}
```

Status: **Resolved in V1.1**

[M-02] Centralization risk

Description:

The current setup of the project grants extensive authority to the owner role, allowing them to control critical functions that influence the core functionality of the system. If the owner account were to be compromised, it could lead to severe vulnerabilities and potential exploitation. Below functions are handled by `onlyOwner`:

- Vesting_Proposal
  - `setTokenAddress()`
  - `makeProposal()`
  - `claimfund()`
  - `discardRunningProposal()`
  - `addSigner()`
  - `changeOwner()`
- Vesting(VC)
  - `SetStartingPoint()`
  - `claim()`
  - `changeOwner()`
- Vesting_Marketing
  - `SetStartingPoint()`
  - `claim()`
  - `changeTokenAdress()`
  - `changeOwner()`

Recommendation:

Explore the implementation of a TimeLock contract as the protocol owner, enabling users to oversee and understand proposed changes before they are executed. Alternatively, consider transferring the admin role to a governance-controlled address, promoting community involvement and transparency in decision-making processes.

Status: **Acknowledged**

[M-03] Missing Functionality to Update and Remove Signers

Reference: Vesting_Proposal

Description:
`OCH_VESTING_PROPOSAL` contract lacks functionality to remove signers once they have been added. After deployment the contract does not provide any means to update or remove the signers. This functionality becomes important when some signer behaves malicious or they lost control of their wallet in event of security breach.

Recommendation:
Consider adding function to allow authorized addresses to update or remove Signers.

Status: **Acknowledged**

[M-04] Return value of `transfer` is not checked

Reference:
1) Vesting_Marketing#L66
2) Vesting(VC)#L66

Description:
In Solidity, when you call the transfer method of an `ERC20` token, it should return a Boolean value indicating success or failure. However, the `claim()` of *OCH_VESTING_MARKETING* contract assumes that this transfer will always succeed and does not check the return value. By not checking the return value, the contract assumes the transfer will never fail, which is not safe. If the transfer does fail (due to a lack of balance, token contract issues, or other reasons), the claim function would still execute and set `lastTimeClaimed` to the current timestamp, potentially leading to a loss of funds or incorrect vesting state without any indication of the failure.

Recommendation:
It is good to add a `require()` statement that checks the return value of token transfers or to use OpenZeppelin's `safeTransfer`/`safeTransferFrom` unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

Status: **Acknowledged**

Low Severity Issues:

[L-01] `changeOwner()` should be 2-step process

Reference:

1) **Vesting_Proposal#L112**
2) **Vesting_Marketing#L74**
3) **Vesting(VC)#L72**

Description:

Lack of two-step procedure for critical operations (like change owner address) leaves them error-prone.

Recommendation:

Implement a two-step owner changing process:

1. The existing owner nominates a new owner using the `setOwner()`
2. The new owner accepts the nomination using an `acceptOwnerNomination()`.
3. After accepting the nomination, the candidate becomes an owner.

```
struct OwnerCandidate {
    bool exists;
    bool accepted;
}
mapping(address => OwnerCandidate) private ownerCandidates;
mapping(address => bool) public owners;

function changeOwner(address _newOwner) public onlyOwner {
    if (_newOwner == address(0)) revert Errors.ZeroAddress();
    ownerCandidates[_newOwner].exists = true;
    ownerCandidates[_newOwner].accepted = false;
    emit Events.AdminNominated(_newOwner);
}

function acceptOwnerNomination() public {
    require(adminCandidates[msg.sender].exists, "No admin nomination found for this
address");
    require(!adminCandidates[msg.sender].accepted, "You have already accepted admin
nomination");

    adminCandidates[msg.sender].accepted = true;
    admins[msg.sender] = true;
    emit Events.NewOwnerAdded(msg.sender);
}
```

Status: **Acknowledged**

[L-02] Missing zero-address and values check in constructors and the setter functions

Reference:

1) **Vesting_Proposal#L43, L50, L54, L83, L112**
2) **Vesting_Marketing#L39, L44, L62, L71, L74**
3) **Vesting(VC)# L39, L44, L72**

Description:

Missing checks for zero-addresses and zero value may lead to unfunctional protocol, if the variable addresses and values are updated incorrectly.

It's noted that all setter functions in the contract utilize the `onlyOwner` modifier, ensuring they can only be called by authorized individuals. However, there exists a potential vulnerability where an owner might inadvertently add `address(0)`. To enhance security, it's advisable to include a check for the zero address and values before assigning addresses and values.

Recommendation:

Consider adding zero-address and values checks in the constructors and setter functions. For zero-address the recommended approach outlined in issue [G-07] can be utilized.

Status: **Acknowledged**

[L-03] Missing event for critical functions

Reference:

1) **Vesting_Proposal**
2) **Vesting_Marketing**
3) **Vesting(VC)**

Description:

Functions that change critical contract parameters/addresses/state should emit events so that users and other privileged roles can detect upcoming changes (by off-chain monitoring of events). Here any of the functions are not emitting any events.

Status: **Acknowledged**

[L-04] No check for contract balance before making proposal and claiming funds.

Reference:

1) **Vesting_Proposal#L54-L62**
2) **Vesting_Proposal#L83-L93**

Description:

The `makeProposal()` allows the owner to create a new proposal to withdraw a specified amount of tokens without verifying if the contract has a sufficient balance of tokens to cover the withdrawal amount. Same thing happens with `claimfund()` as well which is not checking contract balance before transfer funds.

Due to this, a proposal can be created for more tokens than the contract actually holds. If it approved, the `claimfund()` could fail when attempting to transfer more tokens than available, leading to a locked state. Moreover, users may vote on and approve a proposal that cannot be executed, wasting resources.

Recommendation:

Check the contract's token balance before setting the withdrawal amount in `makeProposal()` and before transferring tokens in `claimfund()`.

Status: **Resolved in 1.2**

[L-05] Lack of pause/unpause functionality

Description:

The contract lacks upgradeability and pause functionality, which means that if a critical bug or security vulnerability is discovered, there is no way to halt operations or apply a fix without deploying a new contract and migrating the state and funds. Due to this there will be inability to respond quickly to discovered vulnerabilities, potentially leading to loss of funds or other critical issues. And no way to stop potential malicious activity or accidental transactions during an emergency.

Recommendation:

Use Openzeppelin's `pausable` library.

Status: **Acknowledged**

Informational Issues

[NC-01] Avoid hardcoding values

Reference:

1) **Vesting_Proposal#L46, L84, L103**
2) **Vesting_Marketing#L57, L63**
3) **Vesting(VC)#L57, L63**

Recommendation:

Avoid hardcoding values; instead, use variables to facilitate future changes or constant variables if no changes are planned.

Status: Acknowledged

[NC-02] Remove Unused/*Commented code*

Reference: Vesting_Proposal#L108-L110

Status: Acknowledged

[NC-03] Lack of Comments and Documentation

Description:

The contract code provided lacks comments and documentation, which are essential for understanding the purpose, functionality, and expected behavior of functions within the contract. It will cause poor maintainability, as future updates or modifications may unintentionally break functionality due to a lack of understanding of the original code's intent.

Recommendation:

Add NatSpec comments to all functions, describing their purpose, parameters, return values, and any side effects or requirements.

Status: Acknowledged

Gas Optimization

[G-01] Unnecessary incrementing values

Reference:

1) **Vesting_Marketing#L58**
2) **Vesting(VC)#L58**

Description:

In the *calculateReleaseToken()*, the variable *returnAmount* is initialized to zero and is only assigned a value once within the function.

```
returnAmount += (time * perSecPercent);
```

Since returnAmount is initialized to zero at the start of the function and not modified anywhere else before this line, the += operator is unnecessary.

Recommendation:

Replace += with =

```
returnAmount = (time * perSecPercent);
```

Status: **Acknowledged**

[G-02] Should not perform a lookup for *<array>.length* within each iteration of a for-loop

Reference:

1) **Vesting_Proposal#L66**
2) **Vesting_Proposal#L72**

Recommendation:

Optimizing the loop by storing the array's length in a variable before entering it can significantly reduce gas consumption. In scenarios where the length is fetched from memory, this approach can save approximately 3 gas per iteration. Thus, it's recommended to cache the array's length in a variable and use this variable within the loop for better efficiency.

Status: **Acknowledged**

[G-03] Use the *constant* keyword for unchanging variables

Reference: Vesting_Proposal#L94-L100

Description:

The *discardRunningProposal()* can be executed regardless of whether a proposal is currently active. This means that the owner can call this function at any time, even if there is no proposal to discard, leading to unnecessary gas consumption and state changes that do not reflect any meaningful action.

Recommendation:

Add a check to ensure that there is an active proposal before allowing the state to be reset:

```
require(isProposalActive, "No active proposal to discard");
```

Status: Acknowledged

[G-04] Unnecessary checks and operations that could be optimized

Reference: Vesting_Proposal#L64-L81

Description:

1) The *voteForProposal()* uses a linear search to check if *msg.sender* has already voted, which is inefficient for large arrays.

```
if (msg.sender == VotedForProposal[i])
```

2) Another linear search is used to check if *msg.sender* is in the *multisigner[]*, which is also inefficient and could be costly in terms of gas if the array grows large.

```
if (msg.sender == multisigner[i]) {
```

Recommendation:

1) Replace the array for VotedForProposal with a mapping to track whether an address has voted, allowing for constant-time lookups.

```
mapping(address => uint256) public hasVoted;
```

2) Use a mapping for multisigner to quickly verify if an address is authorized to vote, avoiding the need for a loop.

```
mapping(address => uint256) public isMultisigner;
```

Note: uint256 is recommended instead of bool. 0 and 1 will be more gas efficient instead of true and false

Status: Acknowledged

[G-05] Avoid initializing variables to default values

Reference:

1) Vesting_Proposal#L66, L72

Description:

Explicitly initializing a variable with its default value, such as `0` for `uint`, `false` for `bool`, or `address(0)` for address when it's not set/initialized, is considered an anti-pattern and results in unnecessary gas consumption.

Status: Acknowledged

Closing Summary

In this audit, we examined the NFTFN's smart contract with our framework, and we discovered several High, Medium, Low, Informational and Gas Optimizations 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: 9

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: secureverse@protonmail.com