Cyberscope

Audit Report

One Rich

Aug 2023

Network	BSC
Address	0x1767d519A11eE0A0073BE1941125Dc459F655B2E
Audited by	© cyberscope

Table of Contents

Review

ONERICHGROUP (RICHONE)

Contract Name	ONERICHGROUP
Compiler Version	v0.8.0+commit.c7dfd78e
Optimization	200 runs
Explorer	https://bscscan.com/address/0xdb92be5d6ef6136c3e8d54e161a10e83e4f4a113
Address	0xdb92be5d6ef6136c3e8d54e161a10e83e4f4a113
Network	BSC
Symbol	RICHONE

ONERICHGROUP (RICHTWO)

Contract Name	ONERICHGROUP
Compiler Version	v0.8.0+commit.c7dfd78e
Optimization	200 runs
Explorer	https://bscscan.com/address/0xae2650dfd2b0d435fc395eedf2fd14e30af9a354
Address	0xae2650dfd2b0d435fc395eedf2fd14e30af9a354
Network	BSC
Symbol	RICHTWO

ONERICHGROUP (RICHTHREE)

Contract Name	ONERICHGROUP
Compiler Version	v0.8.0+commit.c7dfd78e
Optimization	200 runs
Explorer	https://bscscan.com/address/0x909a73b84c5d394d8ffbf1af87a6686dbdd814ec
Address	0x909a73b84c5d394d8ffbf1af87a6686dbdd814ec
Network	BSC
Symbol	RICHTHREE

ONERICHGROUP (RICHFOUR)

Contract Name	ONERICHGROUP
Compiler Version	v0.8.0+commit.c7dfd78e
Optimization	200 runs
Explorer	https://bscscan.com/address/0x891a9b49c02bd3c87b89e7d0fad209a4f5f7f440
Address	0x891a9b49c02bd3c87b89e7d0fad209a4f5f7f440
Network	BSC
Symbol	RICHFOUR

ONERICH GROUP (ORG)

Contract Name	ORG
Compiler Version	v0.8.21+commit.d9974bed
Optimization	200 runs
Explorer	https://bscscan.com/address/0x1767d519a11ee0a0073be1941125dc459f655b2e
Address	0x1767d519a11ee0a0073be1941125dc459f655b2e
Network	BSC
Symbol	ORG
Decimals	18
Total Supply	200,072

Audit Updates

Initial Audit	23 Jul 2023 https://github.com/cyberscope-io/audits/blob/main/org/v1/audit.pdf
Corrected Phase 2	11 Aug 2023

Source Files

Filename	SHA256
ORG.sol	f6c5963c7c10ddfeabc19f4566cc24405e1df506e21de92b5129fbe35d7dd32c

Overview

The ORG contract is an Ethereum-based smart contract that integrates functionalities of the ERC20 token standard with unique features tailored for staking Non-Fungible Tokens (NFTs) to earn rewards. Built upon the robust foundation of OpenZeppelin's libraries, the contract offers a blend of traditional token operations and innovative staking mechanisms.

Staking Process

Users can stake their NFTs in the contract to participate in the reward distribution, by calling the `stake` function. When staking, a user specifies an `Upline` , which is a referral or upline address, the `tokenId` of the NFT they wish to stake, the type of the NFT (`nft`), and the `pid` which determines the reward rate and timelock for the staked NFT. The NFT is then transferred from the user's address to the contract, ensuring its locked state. The staking details, including the user's address, upline, and other parameters, are logged in the contract's state.

Unstaking Process

Users can `unstake` their NFT by invoking the `unstake` function, providing the `tokenId` of their staked NFT as a parameter. The function also accepts a boolean parameter `is_emergency` . If `is_emergency` is set to `false` , any pending rewards are claimed before the NFT is returned. However, if `is_emergency` is set to `true` , the unstaking process bypasses the reward claim, ensuring a swift return of the NFT without any rewards. After these processes, the NFT is transferred back to the user's address, and the staking data associated with that NFT is reset in the contract.

Claiming Staking Rewards

Users can claim their staking rewards without unstaking their NFTs, by calling the `claim_staking_reward` function. The `claim_staking_reward function` calculates the pending rewards based on the staking duration, the value of the staked NFT, and other parameters. The rewards are then minted and transferred to the staker. If the staker had specified an upline during the staking process, a percentage of the rewards, which are defined by the `config_referral_percent` , is also minted and sent to the upline's address.

Emergency Withdrawal

The `unstake` function has been designed with a built-in `emergency` mechanism. By setting the `is_emergency` flag to `true` when calling the `unstake` function, users can retrieve their staked NFTs without claiming any pending rewards.

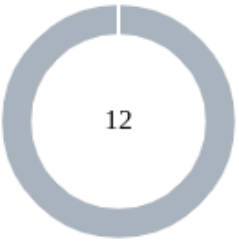
NFTs

The files `nft-one-zephyr.sol` , `nft-two-claritya.sol` , `nft-three-diamore.sol` , and `nft-four-prismora.sol` consist of NFT implementations that are largely similar in their code structure and functionality. However, there are some distinct differences in specific parameters that define the properties of the respective NFTs:

1. ONERICHGROUP (ONEWO):
    - `maxSupply` : The maximum number of tokens that can be minted is set to 50,000.
    - `startId` : The starting ID for the tokens is 1.
    - `mintPrice` : The price to mint a token is set at 100e18.
2. ONERICHGROUP (RICHTWO)I:
    - `maxSupply` : The maximum number of tokens that can be minted is limited to 10,000.
    - `startId` : The starting ID for the tokens is 50,001.
    - `mintPrice` : The price to mint a token is set at 1000e18.
3. ONERICHGROUP (RICHTHREE):
    - `maxSupply` : The maximum number of tokens that can be minted is capped at 1,000.
    - `startId` : The starting ID for the tokens begins at 60,001.
    - `mintPrice` : The price to mint a token is set at 10,000e18.
4. ONERICHGROUP (RICHFOUR):
    - `maxSupply` : The maximum number of tokens that can be minted is restricted to 250.
    - `startId` : The starting ID for the tokens is 61,001.
    - `mintPrice` : The price to mint a token is set at 100,000e18.

These differences in parameters, particularly in `maxSupply`, `startId`, and `mintPrice`, give each NFT implementation its unique characteristics, even though the overarching code structure remains consistent across the files.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	10	2	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	UMP	Unusual Minting Process	Acknowledged
●	IRM	Incorrect Reward Mechanism	Acknowledged
●	MUR	Missing Upline Reset	Unresolved
●	MU	Modifiers Usage	Unresolved
●	TUU	Time Units Usage	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

UMP - Unusual Minting Process

Criticality	Minor / Informative
Location	ONERICHGROUP.sol#L606
Status	Acknowledged

Description

The contract utilizes a minting function that requires users to specify a unique token ID (`_id`) when minting a new NFT. In common NFT minting practices, the token ID is automatically incremented by one each time a new NFT is minted, ensuring a sequential and predictable pattern. However, in the provided contract, users are expected to provide a random, un-minted token ID, which can lead to potential challenges. Users may find it difficult to determine an unused token ID, especially as the total supply grows. This approach complicates the minting process and may deter users from interacting with the contract due to the added complexity and potential for errors.

```
 function mint(uint256 _id) public   {
    if(minted[_id]) return;
    require(_id < startId + maxSupply);
    require(_id >= startId );

    if(msg.sender != owner()) {

IERC20(USDT).transferFrom(address(msg.sender),owner(),mintPrice);
    }

    minted[_id] = true;
    _safeMint(msg.sender, _id);
  }
```

Recommendation

It is recommended to modify the minting function to automatically generate a sequential token ID for each new NFT minted. This can be achieved by maintaining a state variable that tracks the last minted token ID and incrementing it with each mint operation. This change will simplify the user experience, reduce the potential for errors, and align the contract with common NFT minting practices.

Team Update

The team has acknowledged the finding.

IRM - Incorrect Reward Mechanism

Criticality	Minor / Informative
Location	ORG.sol#L417,419
Status	Acknowledged

Description

The contract is designed to compute pending rewards for staked NFTs using the `pendingreward` function. In order to simplify the process, lets assume that all the users will unstake their NFTs after 1 year. The sum of all the rewards will equal to the sum of claiming all the staked NFTs. This equals to unstaking the `YearlyDistributionStaking` amount. This scenario leads to flawed calculations, for example:

```
uint256 persecond = (((YearlyDistributionStaking * unMint *
nft_logs[tokenId].nft)/100)/secondperyears)/CountLockedNftFloor;
```

These can lead to users receiving incorrect reward amounts, as it is shown in the subsequent lines:

```
function pendingreward(uint256 tokenId) public view returns(uint256) {
        if(nft_logs[tokenId].addr==address(0)) return 0;

        // 2851711026615969.5 = 5*1800000e18/100/3.156e7
        uint256 persecond = (((YearlyDistributionStaking * unMint *
nft_logs[tokenId].nft)/100)/secondperyears)/CountLockedNftFloor;

        // 9e+22 = 3.156e7 * 2851711026615969.5
        uint256 diverent = (block.timestamp -
nft_logs[tokenId].lastclaim) * persecond;

        // 1.71e+24 = 1.8e+24 - 9e+22
        // 1_710_000e18 = 1_800_000e18 - 90000e18
        uint256 unMintAfterDiverent = unMint - diverent ;
//Math.min(unmintbefore,unmintafter)

        // 2709125475285171 = 5 * 1.71e+24 / 100 / 3.156e7
        uint256 floor_persecond  = (((YearlyDistributionStaking *
unMintAfterDiverent * nft_logs[tokenId].nft) / 100) /
secondperyears)/CountLockedNftFloor;

        // 8.55e+22 = 3.156e7 * 2709125475285171
        // 85_500e+18
        uint256 AvailableReward  = (block.timestamp -
nft_logs[tokenId].lastclaim) * floor_persecond;

        // 31560000000000000000 = 3.156e7 * 1e10 * 100
        uint256 MaxReward = (block.timestamp -
nft_logs[tokenId].lastclaim) * nft_logs[tokenId].rewardpersecond *
nft_logs[tokenId].nft;

        if(MaxReward>AvailableReward)MaxReward = AvailableReward;
        return  MaxReward ;
    }
```

As a result the actual reward computed after 1 year is `85_500e+18` . This value deviates from the intended `10%` of the `unmint` amount as described in the documentation requirements. Such discrepancies can lead to potential trust issues among stakeholders, misalignment with the project's tokenomics, and unintended financial implications for users.

Additionally, the `MaxReward` variable, as computed in the function, results in an exceptionally large number. This value is inconsistent with the provided documentation which is 10% from `Unmint` Supply, indicating a discrepancy between the intended and actual behavior of the contract.

Recommendation

The team is advised to:

1. Refactor the `pendingreward` function to guarantee its calculations align with the provided documentation, ensuring transparency and preserving stakeholder trust.
2. Re-evaluate the calculation of the `MaxReward` variable to ensure it aligns with the provided documentation. Specifically, the maximum reward should be capped at 10% of the `Unmint` Supply. Implementing a clear and explicit cap in the code will ensure consistency with the intended behavior.

Team Update

The team has acknowledged that this is not a security issue and states:

"*With this distribution model , user must claim periodicaly , because after someone claim , reward pool will reducted , the more often you claim rewards, the rewards you get will be greater than those who claim rewards over a longer period. MaxReward : Use for set maximum APR if price ORG rise more than expected, with target staking APR 10-20% / Year , distribution of reward will lower than maximum target (10% Mint).*"

MUR - Missing Upline Reset

Criticality	Minor / Informative
Location	ORG.sol#L478
Status	Unresolved

Description

The contract resets certain data fields during the `unstake` function. Specifically, within the `unstake` function, the following data fields are reset: `nftdata.addr` , `nftdata.timestamp` , and `nftdata.lastclaim` . However, the `nftdata.upline` field is not being reset. This implies that once an upline address is set to the `nftdata.upline` value (during the `stake` function), this upline address will remain unchanged indefinitely, even after the `unstake` process.

```
function unstake(uint256 tokenId ,bool is_emergency) public {
        require(tokenId>0,"Require tokenId");
        if(!is_emergency){
        claim_staking_reward(tokenId);
        }
        nft_log storage nftdata = nft_logs[tokenId];
        if(nftdata.timestamp<=block.timestamp &&
nftdata.addr==msg.sender){

IERC721(NFT[nftdata.nft]).safeTransferFrom(address(this),msg.sender,
tokenId);

            emit Unstake(nftdata.addr,nftdata.addr ,   tokenId,
nftdata.pid);


            //reset data after unstaking
            nftdata.addr = address(0);
            nftdata.timestamp = 0;
            nftdata.lastclaim = 0;
            ...

        }

    }
```

Recommendation

It is recommended to reset the value of `nftdata.upline` in the appropriate segment of the `unstake` function. If the intended functionality is to clear the `nftdata.upline` value during the unstaking process, this action should be implemented. On the other hand, if the design intention is to reset the `nftdata.upline value` only when the `emergency` flag is set to `true`, then this functionality should be considered for implementation in the specific segment of the code where the `is_emergency` variable is set to `true`.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	ORG.sol#L528,562
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(tokenId>0,"Require tokenId");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	ORG.sol#L269,275
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256[]
config_timelock=[7890000,15780000,31560000,47340000,63120000,94680000];

uint256 public secondperyears = 31560000;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	ORG.sol#L320
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
max_reward_persecond
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	ONERICHGROUP.sol#L577,578,579,581
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 public maxSupply = 50000
uint256 public startId = 1
uint256 public mintPrice = 100e18
address USDT = 0x55d398326f99059fF775485246999027B3197955
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	ORG.sol#L264,268,269,270,272,273,274,275,276,277,284,295,304,308,314,339,351,362,408,411,432,452,465,526ONERICHGROUP.sol#L336,581,596,602,633,637
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
mapping(uint256 => address) public NFT
uint256[] public max_reward_persecond
uint256[]
config_timelock=[7890000,15780000,31560000,47340000,63120000,94680000]
uint256 constant   config_referral_percent = 10
uint256 public LastUseDeFi
uint256 constant   YearlyDistributionDeFi = 5
uint256 constant   YearlyDistributionStaking = 5
uint256 constant   secondperyears = 31560000
address public DeFiContract
bool    public DeFiIsFix = false

...
```

```
bytes memory _data
address USDT = 0x55d398326f99059fF775485246999027B3197955

function update_pool(address _pool) public {
    if(msg.sender == owner() && _pool != address(0)) {
        pool = _pool;
        }
    }
address _pool
uint256 _id
string memory _newBaseURI
string memory _newBaseExtension
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	ONERICHGROUP.sol#L73,86,110,117,121,129,137,150,154,165,169,180,284,387
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function toHexString(uint256 value) internal pure returns (string
memory) {
        if (value == 0) {
            return "0x00";
        }
        uint256 temp = value;
        uint256 length = 0;
        while (temp != 0) {
            length++;
            temp >>= 8;
        }
        return toHexString(value, length);
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	ONERICHGROUP.sol#L104,191,429
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            size := extcodesize(account)
        }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }

assembly {
                        revert(add(32, reason), mload(reason))
                    }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	ONERICHGROUP.sol#L3
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	ONERICHGROUP.sol#L608
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(USDT).transferFrom(address(msg.sender),owner(),mintPrice)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC721Receiver	Interface			
	onERC721Received	External	✓	-
IERC165	Interface			
	supportsInterface	External		-
IERC721	Interface	IERC165		
	balanceOf	External		-
	ownerOf	External		-
	safeTransferFrom	External	✓	-
	safeTransferFrom	External	✓	-
	transferFrom	External	✓	-
	approve	External	✓	-
	setApprovalForAll	External	✓	-
	getApproved	External		-
	isApprovedForAll	External		-
IERC20	Interface			

	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IERC20Metadata	Interface	IERC20		
	name	External		-
	symbol	External		-
	decimals	External		-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
ERC20	Implementation	Context, IERC20, IERC20Metadata		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-

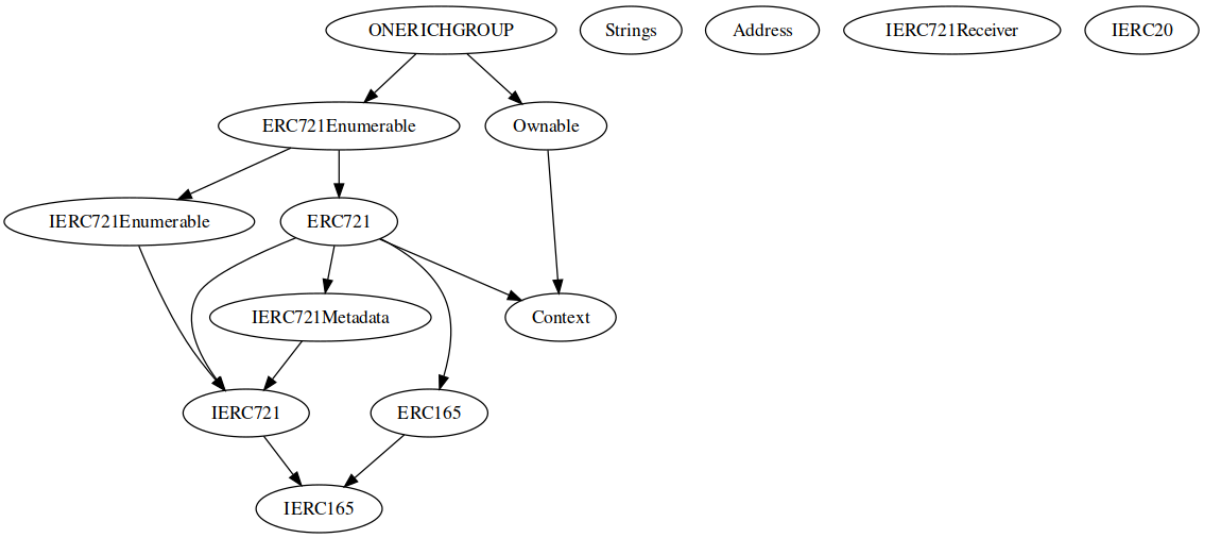
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	_mint	Internal	✓	
	_burn	Internal	✓	
	_approve	Internal	✓	
	_spendAllowance	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
ERC20Burnable	Implementation	Context, ERC20		
	burn	Public	✓	-
	burnFrom	Public	✓	-
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		

	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
ORG	Implementation	ERC20, Ownable, ERC20Burnable, IERC721Receiver		
	onERC721Received	External		-
		Public	✓	ERC20
	_limit_mint	Internal	✓	
	maxSupply	Public		-
	reconfig_reward	Public	✓	onlyOwner
	stake	Public	✓	-
	stakings_length	Public		-
	user_stakings_length	Public		-
	pendingreward	Public		-
	claim_staking_reward	Public	✓	-
	configinfo_reward	Public		-
	unstake	Public	✓	-
	defiPool	Public		-
	setContract	Public	✓	onlyOwner
	SetFixDefi	Public	✓	onlyOwner
	moveToDefi	Public	✓	-

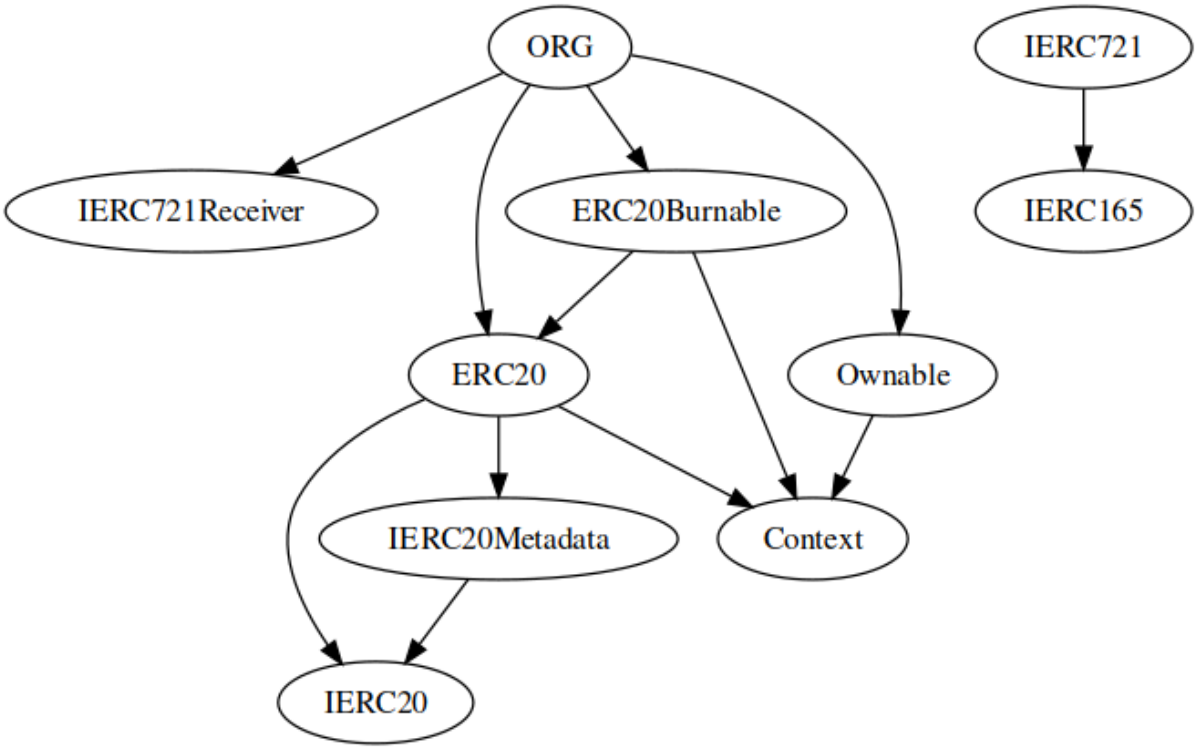
ONERICHGROUP	Implementation	ERC721Enumerable, Ownable		
		Public	✓	ERC721
	_baseURI	Internal		
	update_pool	Public	✓	-
	mint	Public	✓	-
	tokenURI	Public		-
	setBaseURI	Public	✓	onlyOwner

Inheritance Graph

ONERICHGROUP

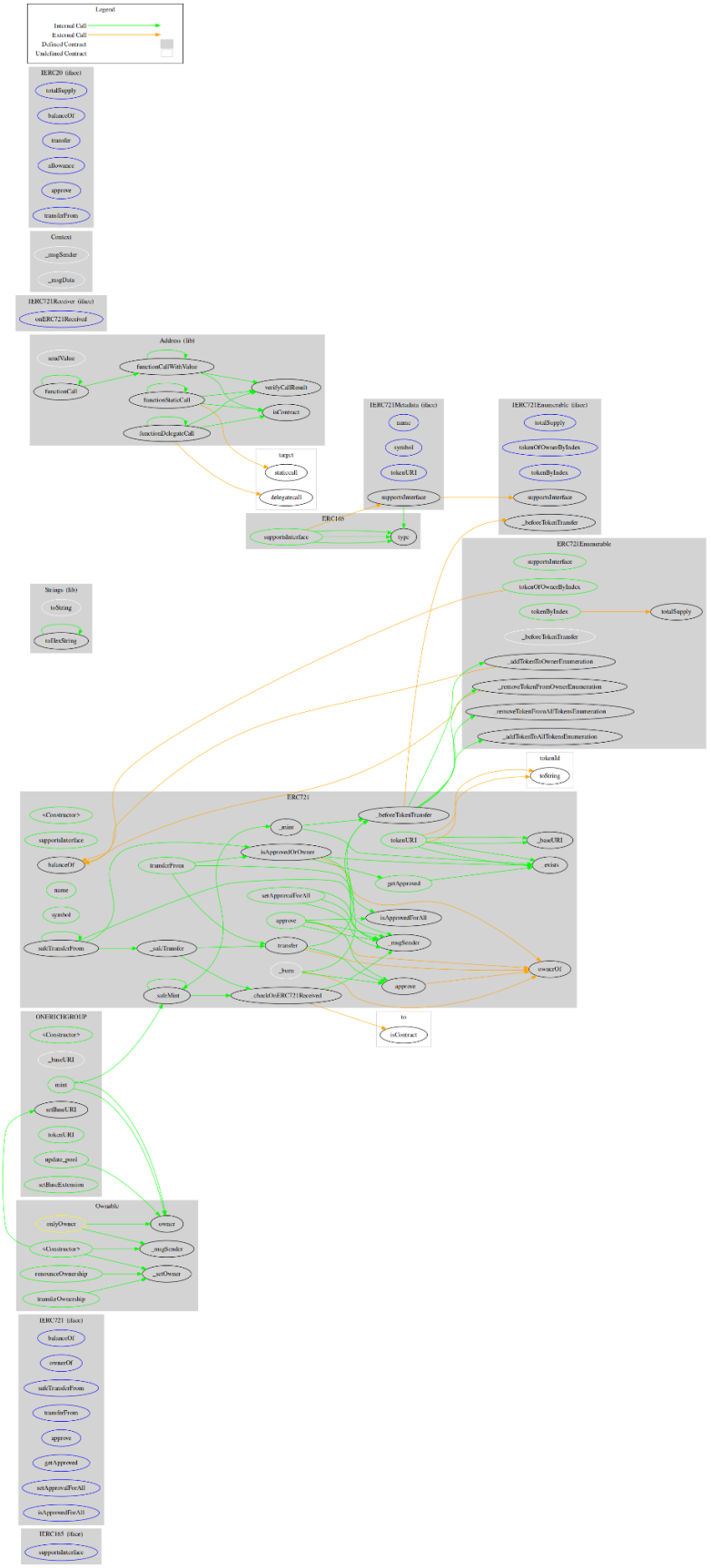


ONERICH GROUP (ORG)

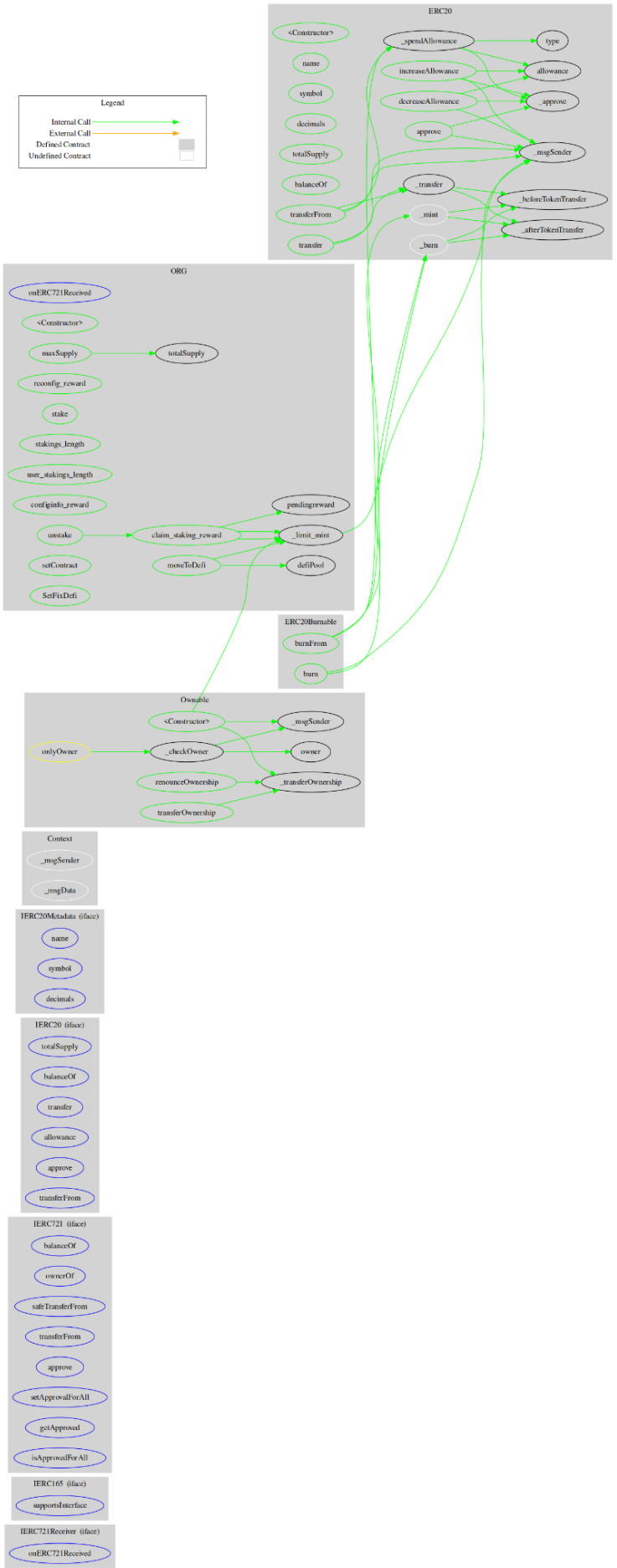


Flow Graph

ONERICHGROUP



ONERICH GROUP (ORG)



Summary

The One Rich contract suite is divided into two main components. The ONERICHGROUP which consists of RICHONE, RICHTWO, RICHTRHEE, and RICHFOR is responsible for implementing the NFT functionalities. On the other hand, the ONERICH GROUP (ORG) implements token, staking, and rewards mechanisms. This audit delves into the security aspects, evaluates business logic concerns, and suggests potential improvements for both components.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io