Audit Report

One Rich

July 2023

Commit	0bf35283dca75e1b3c8281a6565f0ef99bf78546
Repository	https://github.com/onerichgroup/smartcontract/blob/main/org/org.sol
Audited by	© cyberscope

Table of Contents

Review

Repository	https://github.com/onerichgroup/smartcontract/blob/main/org/org.sol
Commit	0bf35283dca75e1b3c8281a6565f0ef99bf78546

Audit Updates

Initial Audit	23 Jul 2023

Source Files

Filename	SHA256
org.sol	d0f3bf749bc3a427de17e466ec42e6a8e0e18af811577a9d4b13a64f1bc091e6
nft-two-claritya.sol	9fb51c4fef64224f1d30d1118e7047623e3729e48e4dc4b75f072952cb2ae5b2
nft-three-diamore.sol	bfb16d926e9d83978fe33bf04bfbd6fed4afb1ee23eac6712ad8df7bf87f50af
nft-one-zephyr.sol	4f8b0761397deb7a5ffb75f30a1fc06aba0d8e7566f59b73a785e37f592b62ac
nft-four-prismora.sol	98dc916ceec2b34e0423cd0015ea59b956f0e392583346a04190e4c85ae73bf9

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 token 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 calling the `unstake` function with the `tokenId` of their staked NFT as a parameter. Before the NFT is returned, any pending rewards are claimed. The NFT is then transferred back to the user's address, and the staking data for 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

In exceptional scenarios, users can retrieve their staked NFTs without claiming the rewards using the `emergency` function. This function transfers the staked NFT back to the user's address without any reward distribution. It's designed as a safety mechanism, ensuring users can always access their assets.

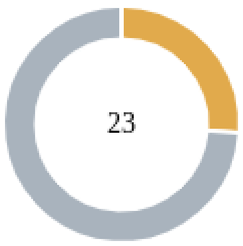
NFTs

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. nft-one-zephyr.sol:
   - `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. nft-two-claritya.sol:
   - `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. nft-three-diamore.sol:
   - `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. nft-four-prismora.sol:
   - `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	6
●	Minor / Informative	17

	Severity	Unresolved	Acknowledged	Resolved	Other
🟥	Critical	0	0	0	0
🟧	Medium	6	0	0	0
●	Minor / Informative	17	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	URI	Upline Reward Inflation	Unresolved
●	IRM	Incorrect Reward Mechanism	Unresolved
●	PCR	Potential Circular Referral	Unresolved
●	UMP	Unusual Minting Process	Unresolved
●	MC	Missing Check	Unresolved
●	SCD	Supply Configuration Discrepancy	Unresolved
●	RAU	Redundant Array Usage	Unresolved
●	CR	Code Repetition	Unresolved
●	MU	Modifiers Usage	Unresolved
●	TUU	Time Units Usage	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSK	Redundant Storage Keyword	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
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

URI - Upline Reward Inflation

Criticality	Medium
Location	org.sol#L503
Status	Unresolved

Description

The contract is over-rewarding in the `claim_staking_reward` function. Beyond the primary staking rewards (`pending`), it mints an extra 10% of the `pending` rewards to the `upline` address, as demonstrated by the code:

```
uint256 affReward  = pending.mul(config_referral_percent).div(100);
_limit_mint(nftdata.upline, affReward);
```

This means that for every reward claim, the total number of tokens minted is increased by more than the actual staking rewards (pending), leading to potential inflation of the token supply. Over time, this could dilute the value of the tokens and may not align with the intended tokenomics or the expectations of the stakeholders.

Recommendation

The team is advised to review the tokenomics and the planned reward distribution strategy to ensure that the extra 10% minting to the upline address meets the project's objectives. All supplementary minting for upline addresses must be accurately accounted for.

IRM - Incorrect Reward Mechanism

Criticality	Medium
Location	org.sol#L482,503
Status	Unresolved

Description

The contract is designed to compute pending rewards for staked NFTs using the `pendingreward` function. In order to simplify the process, let's assume that all the users will unstake their NFTs after 1 year. The sum of all the rewards will equal 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.mul(unMint).mul(CountLockedNftFloor).div(100).d
iv(secondperyears).div(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) {
        nft_log storage nftdata = nft_logs[tokenId];
        if(nftdata.addr==address(0)) return 0;

        // 2851711026615969.5 = 5*1800000e18/100/3.156e7
        uint256 persecond =
YearlyDistributionStaking.mul(unMint).mul(CountLockedNftFloor).div(100).d
iv(secondperyears).div(CountLockedNftFloor);

        // 9e+22 = 3.156e7 * 2851711026615969.5
        uint256 diverent =
block.timestamp.sub(nftdata.lastclaim).mul(persecond);

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

        // 2709125475285171 = 5 * 1.71e+24 / 100 / 3.156e7
        uint256 floor_persecond  =
YearlyDistributionStaking.mul(unMintAfterDiverent).mul(CountLockedNftFloo
r).div(100).div(secondperyears).div(CountLockedNftFloor);

        // 8.55e+22 = 3.156e7 * 2709125475285171
        // 85_500e+18
        uint256 AvailableReward  =
block.timestamp.sub(nftdata.lastclaim).mul(floor_persecond);

        // 315600000000000000000 = 3.156e7 * 1e10 * 100
        uint256 MaxReward =
block.timestamp.sub(nftdata.lastclaim).mul(nftdata.rewardpersecond).mul(n
ftdata.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.

PCR - Potential Circular Referral

Criticality	Medium
Location	org.sol#L431
Status	Unresolved

Description

The contract allows users to stake NFTs and specify an `upline` address through the `stake` function. However, the code does not prevent users from passing their own address (`msg.sender`) as the `upline` address. This could lead to potential misuse or unintended behaviors, as users might be incentivized to set themselves as their own upline, bypassing intended referral or reward mechanisms.

```
    function stake( address Upline,uint256 tokenId,uint256
pid) public {
        require(tokenId>0,"Require nft id");
        require(NFT[nft] != address(0),"Require valid nft");
        ...
        if(upline[msg.sender] != address(0)) Upline =
upline[msg.sender];
        else upline[msg.sender] = Upline;

        ...
    }
```

Recommendation

It is recommended to add a validation check within the `stake` function to ensure that the `Upline` address provided is not equal to the `msg.sender` address. This can be achieved with a simple require statement, such as `require(Upline != msg.sender, "Comment")`. Implementing this check will ensure that users cannot set themselves as their own upline, aligning the function's behavior with its likely intended purpose.

UMP - Unusual Minting Process

Criticality	Medium
Location	nft-one-zephyr.sol#L606nft-two-claritya.sol#L606nft-three-diamore.sol#L606nft-four-prismora.sol#L606
Status	Unresolved

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.

MC - Missing Check

Criticality	Medium
Location	contracts/org.sol#L431
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, within the `stake` function, the parameter `pid` is passed. However, there is no validation in place to ensure that the provided `pid` is within an acceptable range. This omission can lead to unintended behavior or misuse of the function.

```
    function stake( address Upline,uint256 tokenId,uint256 nft,uint256
pid) public {
        require(tokenId>0,"Require nft id");
        require(NFT[nft] != address(0),"Require valid nft");
        ...
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to implement checks within the `stake` function to validate the provided `pid`. Ensure that the `pid` falls within the expected range of valid ID values.

SCD - Supply Configuration Discrepancy

Criticality	Medium
Location	org.sol#L394
Status	Unresolved

Description

The contract is currently configured with an initial max supply of `20,000,000` ORG tokens, as indicated by the `unMint` variable set to `20e24`. However, the initial requirements provided specify that the intended max supply should be `2,000,000` ORG. This discrepancy can lead to potential issues, such as unintended inflation, dilution of token value, and misalignment with project goals or tokenomics.

```
constructor() ERC20("ONERICH GROUP", "ORG") {

    _limit_mint(msg.sender, 2e24); //initial mint
    ...
  }
```

Recommendation

It is recommended to update the `unMint` variable to reflect the intended max supply of `2,000,000` ORG, which would be represented as `2e24`. Additionally, it would be prudent to review other parts of the contract to ensure that all references to the max supply are consistent and aligned with the intended value.

RAU - Redundant Array Usage

Criticality	Minor / Informative
Location	contracts/org.sol#L396,444
Status	Unresolved

Description

The contract initializes the array `config_timelock` with all elements set to zero. Although this array is used in the contract's code, it has no actual effect on the logic or computation since all its elements are set to zero. Throughout the contract's code, this variable is never updated. Such unused state variables can lead to confusion for developers and auditors, increase the contract's complexity, and potentially waste gas when deploying the contract.

```
config_timelock = [0,0,0,0,0,0];

nftdata.timestamp    = block.timestamp.add(config_timelock[pid]);
```

Recommendation

It is recommended to remove the `config_timelock` state variable from the contract if it serves no functional purpose. If there are future plans to utilize this variable, consider adding comments or documentation to clarify its intended use. Otherwise, eliminating unused variables can streamline the contract, making it more readable and efficient.

CR - Code Repetition

Criticality	Minor / Informative
Location	org.sol#L527,561
Status	Unresolved

Description

The contract contains repetitive code segments. Specifically, the contract has a code repetition between the `unstake` and `emergency` functions. Both functions share almost identical code segments, with the primary differences being the calculation of `claim_staking_reward(tokenId)` in the `unstake` function and the setting of `nftdata.upline = address(0)` in the `emergency` function. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
   function unstake(uint256 tokenId) public {
        require(tokenId>0,"Require tokenId");
        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);
            //reset data after unstaking
            nftdata.addr = address(0);
            nftdata.timestamp = 0;
            nftdata.lastclaim = 0;

            CountLockedNftFloor=CountLockedNftFloor.sub(nftdata.nft);

            stakings.push(staking({
                addr:msg.sender,
                tokenid:tokenId,
                timestamp:block.timestamp,
                pid : nftdata.pid,
                thisIn:false
            }));

            nftlogs[msg.sender].push(nft_users({
                tokenId:tokenId,
                timestamp: nftdata.timestamp,
                thisIn:false
            }));
         }
      }

     function emergency(uint256 tokenId) public {
        ...
           nftdata.upline = address(0);
        ...
         }
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/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	contracts/org.sol#L350
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 public secondperyears = 3.156e7;
```

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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/org.sol#L431,498,527,561,622
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

Specifically, the functions `stake` , `claim_staking_reward` , `unstake` , `emergency` , and `moveToDefi`

```
function stake( address Upline,uint256 tokenId,uint256 nft,uint256 pid)
public {
        require(tokenId>0,"Require nft id");
        require(NFT[nft] != address(0),"Require valid nft");
        nft_log storage nftdata = nft_logs[tokenId];
    ...
    }

 function claim_staking_reward(uint256 tokenId) public  {
 ...
 }

 function unstake(uint256 tokenId) public {
 ...
 }

 function emergency(uint256 tokenId) public {
 ...
 }

 function moveToDefi() public {
 ...
 }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	org.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSK - Redundant Storage Keyword

Criticality	Minor / Informative
Location	org.sol#L483
Status	Unresolved

Description

The contract uses the `storage` keyword in a view function. The `storage` keyword is used to persist data on the contract's storage. View functions are functions that do not modify the state of the contract and do not perform any actions that cost gas (such as sending a transaction). As a result, the use of the `storage` keyword in view functions is redundant.

```
nft_log storage nftdata
```

Recommendation

It is generally considered good practice to avoid using the `storage` keyword in view functions because it is unnecessary and can make the code less readable.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	org.sol#L395,396
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
config_timelock
```

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	org.sol#L345,348,349,350nft-two-claritya.sol#L581,582,583,585nft-three-diamore.sol#L581,582,583,585nft-one-zephyr.sol#L581,582,583,585nft-four-prismora.sol#L581,582,583,585
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 config_referral_percent = 10
uint256 public YearlyDistributionDeFi = 5
uint256 public YearlyDistributionStaking = 5
uint256 public secondperyears = 3.156e7
uint256 public maxSupply = 10000
uint256 public startId = 50001
uint256 public mintPrice = 1000e18
address USDT = 0x55d398326f99059fF775485246999027B3197955
uint256 public maxSupply = 1000
uint256 public startId = 60001
uint256 public mintPrice = 10000e18
uint256 public maxSupply = 50000
uint256 public startId = 1
uint256 public mintPrice = 100e18

...
```

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#L339,343,344,345,347,348,349,351,352,359,370,379,383,389,408,420,431,473,476,498,514,617nft-two-claritya.sol#L340,585,600,606,637,641nft-three-diamore.sol#L340,585,600,606,637,641nft-one-zephyr.sol#L340,585,600,606,637,641nft-four-prismora.sol#L340,585,600,606,637,641
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[] public config_timelock
uint256 public config_referral_percent = 10
uint256 public LastUseDeFi
uint256 public YearlyDistributionDeFi = 5
uint256 public YearlyDistributionStaking = 5
address public DeFiContract
bool    public DeFiIsFix = false

...
```

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	nft-two-claritya.sol#L77,90,114,121,125,133,141,154,158,169,173,184,288,391nft-three-diamore.sol#L77,90,114,121,125,133,141,154,158,169,173,184,288,391nft-one-zephyr.sol#L77,90,114,121,125,133,141,154,158,169,173,184,288,391nft-four-prismora.sol#L77,90,114,121,125,133,141,154,158,169,173,184,288,391
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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	org.sol#L612
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(DeFiIsFix == false,"Unable to change DeFi contract")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	org.sol#L485,486,488,489,601,602,604,605
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 persecond =
YearlyDistributionDeFi.mul(unMint).div(100).div(secondperyears)
uint256 diverent = block.timestamp.sub(LastUseDeFi).mul(persecond)
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	org.sol#L613
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
DeFiContract = defi
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	nft-two-claritya.sol#L108,195,433nft-three-diamore.sol#L108,195,433nft-one-zephyr.sol#L108,195,433nft-four-prismora.sol#L108,195,433
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	org.sol#L2nft-two-claritya.sol#L7nft-three-diamore.sol#L7nft-one-zephyr.sol#L7nft-four-prismora.sol#L7
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	nft-two-claritya.sol#L612nft-three-diamore.sol#L612nft-one-zephyr.sol#L612nft-four-prismora.sol#L612
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.

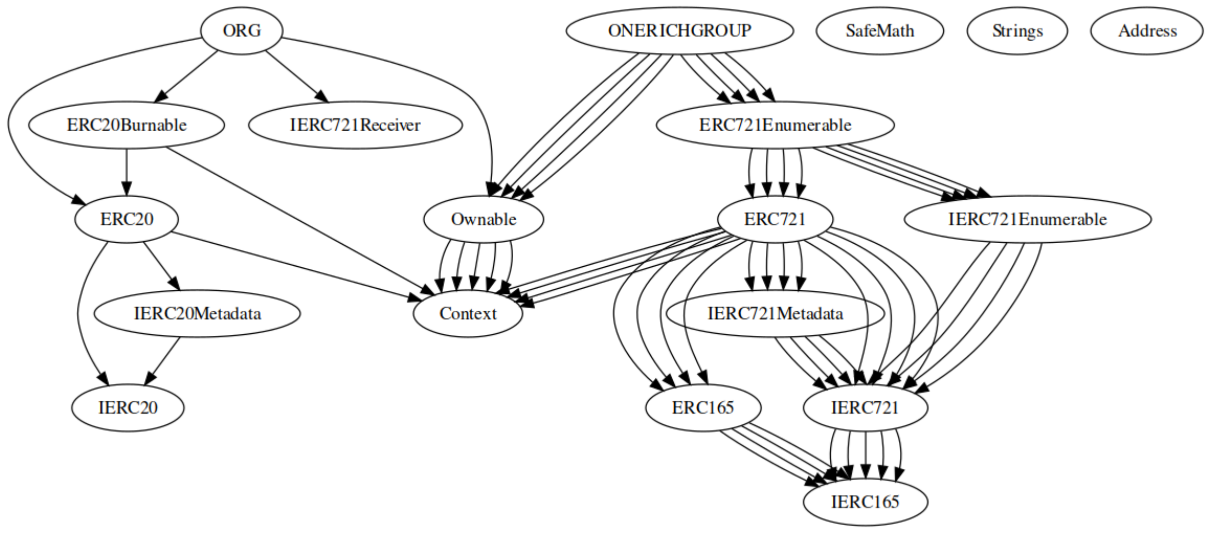
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
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	✓	-
	emergency	Public	✓	-
	defiPool	Public		-
	setContract	Public	✓	onlyOwner
	SetFixDefi	Public	✓	onlyOwner

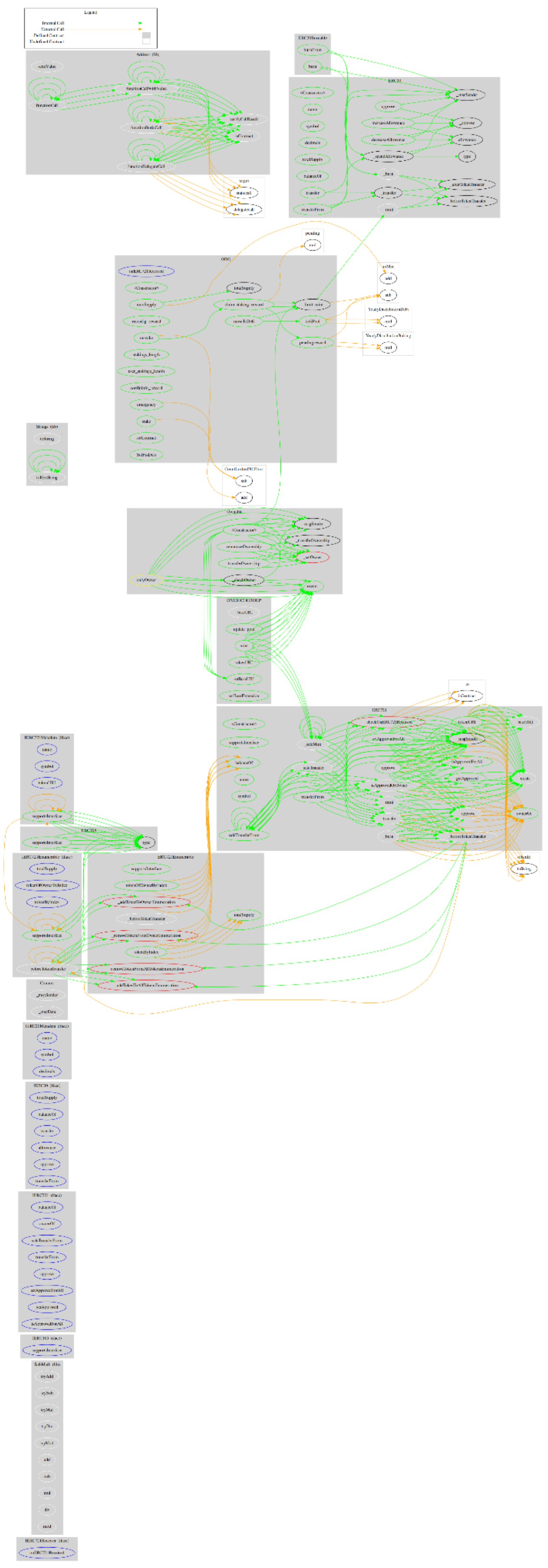
	moveToDefi	Public	✓	-
ONERICHGROUP	Implementation	ERC721Enumerable, Ownable		
		Public	✓	ERC721
	_baseURI	Internal		
	update_pool	Public	✓	-
	mint	Public	✓	-
	tokenURI	Public		-
	setBaseURI	Public	✓	onlyOwner
	setBaseExtension	Public	✓	onlyOwner
ONERICHGROUP	Implementation	ERC721Enumerable, Ownable	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

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

Inheritance Graph



Flow Graph



Summary

One Rich contract implements a token, nft, staking and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

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