Cyberscope

Audit Report

OniToken

May 2024

OniToken	0x0479e8c99cd17D85B30AA07d5e61a65E13610ef2
OniNode	0xec80f0CD2D1c45593572a33a07DfEB05c6Af6C2F
OniIco	0xCb1d47C834e6960b44117dF1471016F0675BB963
OniAffiliate	80400419afe9fd819c17e38c7dc27bf8587ddbe76546c8e4b46606dc073b3853
Audited by	© cyberscope

Table of Contents

Review

Contract Name	OniIco
Exploler	https://bscscan.com/address/0xCb1d47C834e6960b44117dF1471016F0675BB963
Address	0xCb1d47C834e6960b44117dF1471016F0675BB963

Contract Name	OniNode
Exploler	https://bscscan.com/address/0xec80f0CD2D1c45593572a33a07DfEB05c6Af6C2F
Address	0xec80f0CD2D1c45593572a33a07DfEB05c6Af6C2F

Contract Name	OniIco
Exploler	https://bscscan.com/address/0xCb1d47C834e6960b44117dF1471016F0675BB963
Address	0xCb1d47C834e6960b44117dF1471016F0675BB963

Audit Updates

Initial Audit	16 May 2024 https://github.com/cyberscope-io/audits/blob/main/onitoken/v1/audit.pdf
Corrected Phase 2	21 May 2024 https://github.com/cyberscope-io/audits/blob/main/onitoken/v2/audit.pdf

Corrected Phase 3	27 May 2024

Source Files

Filename	SHA256
OniToken.sol	75c5e836a595ea3d3e37a6341b1747bdf8680f119bbdb7415aea813e5f2a2d88
OniNode.sol	09f4f80c7456390d0c4b56e66132f2c1ed7d47f0ae4fe02d1d534cc38bb01cc8
OniIco.sol	b0579b88b01185d33cb3ff752f92c82d642f356207d86058a771d32b4f925732
OniAffiliate.sol	80400419afe9fd819c17e38c7dc27bf8587ddbe76546c8e4b46606dc073b3853

Overview

The Oni ecosystem comprises a suite of interconnected smart contracts designed to create, manage, and promote a custom cryptocurrency and related digital assets. The OniToken contract establishes an ERC20 token with controlled minting and standard token functionalities, ensuring secure and reliable transactions. The OniIco contract facilitates the Initial Coin Offering, enabling users to contribute funds in exchange for tokens and claim them post-ICO, with provisions for referral rewards and flexible fund management. The OniNode contract introduces a dynamic ERC721 NFT system, featuring scalable pricing and a referral program to incentivize participation. Complementing these, the OniAffiliate contract oversees the referral reward system, tracking activities and distributing rewards to promote the ecosystem's growth. Together, these contracts provide a robust framework for token issuance, fundraising, and community-driven promotion within the Oni platform.

OniToken Contract

The OniToken contract implements an ERC20 token implementation. It allows the creation and management of a custom cryptocurrency token with a specified initial supply, name, symbol, and decimal precision. The contract includes functionalities to mint new tokens, which can only be executed by the owner of the contract, ensuring controlled and secure issuance of additional tokens. Users can interact with the token according to standard ERC20 token features, such as transferring tokens, checking balances, and approving allowances.

OniIco contract

The OniIco contract facilitates the Initial Coin Offering (ICO) for the OniToken, enabling users to contribute funds in exchange for OniTokens. It includes mechanisms for secure contributions, referral rewards, and a structured process for claiming tokens after the ICO concludes. Users can contribute by sending funds, and if they have a referrer, the referrer receives a reward. Once the ICO is finished, contributors can claim their allocated tokens based on their contribution amount and the set ICO price. The contract also supports the withdrawal of accumulated funds and allows the owner to update the ICO price and affiliate contract.

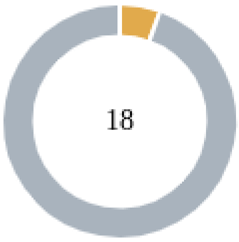
OniNode Contract

The OniNode contract implements an ERC721 token for creating and managing a limited supply of NFTs with dynamic pricing. The contract allows users to mint NFTs, with the mint price increasing after every specified number of mints, ensuring a scalable and fair pricing model. Users can also benefit from a referral system, where referrers earn rewards for referring new buyers. The contract includes mechanisms for updating the maximum supply, base URI, and unrevealed URI for the NFTs, enhancing flexibility and user experience. Additionally, it supports the recovery of ETH and ERC20 tokens by the owner, ensuring efficient fund management and security.

OniAffiliate Contract

The OniAffiliate contract manages the referral reward system for the Oni ecosystem. It tracks sales made through referrals and distributes rewards to referrers based on a specified reward percentage. This contract maintains detailed records of referral activities, including the total sales and rewards per referrer and source. It ensures that only approved sources can initiate referral transactions, enhancing security and integrity. The owner can update the reward percentage and approve or disapprove sources. Additionally, the contract supports the recovery of accumulated ETH and ERC20 tokens, ensuring proper fund management. This setup incentivizes participants to promote the Oni platform, driving growth through a structured and transparent referral progr

Findings Breakdown



Critical	0	
Medium	1	
Minor / Informative	17	

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	17	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	RRM	Referral Reward Misuse	Unresolved
●	APVU	Additional Parameter Variable Usage	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DSU	Duplicate Structs Usage	Unresolved
●	MT	Mints Tokens	Unresolved
●	PBV	Percentage Boundaries Validation	Unresolved
●	RNM	Redundant Non-Reentrant Modifier	Unresolved
●	RRL	Redundant Reward Logic	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved

●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

RRM - Referral Reward Misuse

Criticality	Medium
Location	OniIco.sol#L856 OniNode.sol#L2124 OniAffiliate.sol#L723
Status	Unresolved

Description

The contract is designed to manage referral rewards by allowing users to declare a `referrer` address to transfer the referral reward. However, users can exploit this functionality by declaring a third address that they own, resulting in reduced payments for both the `contribute` and `mint` functions. This misuse allows users to reduce the amount they pay when using the `contribute` and `mint` functionalities, undermining the contract's economic model and potentially leading to financial losses.

```
    function contribute(
        uint256 payAmount,
        address referrer
    ) external payable nonReentrant {
    ...
        oniAffiliate_.saleByReferrer{value: refRewardAmount}(
            caller,
            referrer,
            payAmount
    );
    ...
    }

    function mint(uint256 amount, address referrer) external
payable {
        ...
        oniAffiliate_.saleByReferrer{value: refRewardAmount}(
            caller,
            referrer,
            fundsRequired
        );
    ...
    }

    function saleByReferrer(
        address account,
        address referrer,
        uint256 saleAmount
    ) external payable {
    ...
        uint256 refRewardAmount = (saleAmount *
_refRewardPercent) /
            DENOMINATOR;
        require(msg.value >= refRewardAmount, "insufficient
reward");
        payable(referrer).sendValue(refRewardAmount);
    ...
    }
```

Recommendation

It is recommended to reconsider the intended logic behind the implementation of the referral rewards. One approach could involve developing a more robust referral logic that aligns with the project's overall plans and goals. For instance, instead of sending the rewards directly to the `referrer` address, the contract could store the rewards and distribute them only if the referrer address is also a participant in the system. By implementing these measures, the contract can prevent misuse and maintain the integrity and fairness of the referral reward system.

APVU - Additional Parameter Variable Usage

Criticality	Minor / Informative
Location	OniIco.sol#L856
Status	Unresolved

Description

The contract is designed to accept contributions through the `contribute` function, which takes `payAmount` as a parameter. However, since the function only accepts the native token as payment, it could utilize the `msg.value` directly to set the `payAmount` instead of using an extra variable. This results in additional overhead and can mislead users who might set a `payAmount` less than the `msg.value` , causing confusion and potential errors.

```
    function contribute(
        uint256 payAmount,
        address referrer
    ) external payable nonReentrant {
        require(!_isIcoFinished, "ico finished");
        require(payAmount != 0 && payAmount <= msg.value,
"insufficient funds");
    ...
    }
```

Recommendation

It is recommended to utilize the `msg.value` directly to set and handle the `payAmount` . This simplification will reduce unnecessary overhead and eliminate potential user errors related to mismatched payment amounts.

CR - Code Repetition

Criticality	Minor / Informative
Location	OniNode.sol#L2138,2190
Status	Unresolved

Description

The contract contains repetitive code segments. 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.

Specifically the contract is designed with both a `mint` function and a `requiredFundsForMint` function, each implementing similar logic to calculate the required funds for minting. This results in code repetition, as the same logic is duplicated across both functions. Reusing the existing logic in a centralized manner would enhance maintainability and reduce potential errors or inconsistencies.

```
    function mint(uint256 amount, address referrer) external
payable {
        ...
        for (uint256 i; i < amount; ) {
            uint256 mintIndex = ++totalSupply_;
            _safeMint(caller, mintIndex);

            // calculate required funds
            if (++stepCounter_ > priceChangeStep_) {
                stepCounter_ -= priceChangeStep_;
                mintPrice_ += _priceChangeAmount;
            }
            fundsRequired += mintPrice_;

            unchecked {
                ++i;
            }
        }

        ...

        _mintPrice = mintPrice_;
        stepCounter = stepCounter_;

        emit NodeMinted(caller, referrer, amount);
    }


    function requiredFundsForMint(
        uint256 amount
    ) external view returns (uint256) {
        uint256 fundsRequired;
        uint256 mintPrice_ = _mintPrice;
        uint256 stepCounter_ = stepCounter;
        uint256 priceChangeStep_ = _priceChangeStep;
        for (uint256 i; i < amount; ) {
            // calculate required funds
            if (++stepCounter_ > priceChangeStep_) {
                stepCounter_ -= priceChangeStep_;
                mintPrice_ += _priceChangeAmount;
            }
            fundsRequired += mintPrice_;

            unchecked {
                ++i;
            }
        }

        return fundsRequired;
    }
```

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.

It is recommended to refactor the contract by centralizing the shared logic between the `mint` and `requiredFundsForMint` functions. Specifically, the `mint` function could call the `requiredFundsForMint` function to retrieve the required funds. This change will ensure a single, consistent implementation of the funds calculation logic, making the codebase easier to understand and maintain.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	OniIco.sol#L933,948 OniNode.sol#L2216,2233,2255,2272,2287
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, the contract owner has the authority to set and change the affiliate contract address, ensuring that the affiliate contract used is valid by validating its interface. Additionally, the owner can set the ICO price, which is a crucial parameter as the `claimTokens` function relies on this value for correct calculations. Additionally, the owner can update the maximum supply of NFTs, adjust the minting price, update the price change configuration, and modify the metadata URL.

```
    function updateAffiliate(address affiliate) external
onlyOwner {
        // validate if affiliate contract has proper interface
        IOniAffiliate(affiliate).refRewardPercent();
        _oniAffiliate = affiliate;

        emit AffiliateUpdated(affiliate);
    }

    function updateIcoPrice(uint256 price) external onlyOwner {
        require(price > 0, "invalid price");
        _icoPrice = price;

        emit IcoPriceUpdated(price);
    }

    function updateMintPrice(uint256 price) external onlyOwner
{
        require(_mintPrice != price, "nothing changed");
        _mintPrice = price;
        stepCounter = 0;

        emit MintPriceUpdated(price);
    }

    function updatePriceChangeConf(
        uint256 priceChange,
        uint256 changeStep
    ) external onlyOwner {
        require(changeStep > 0, "invalid change step");
        _priceChangeAmount = priceChange;
        _priceChangeStep = changeStep;
    }

    function updateMaxSupply(uint256 value) external onlyOwner
{
        require(value >= totalSupply(), "already minted more");
        _maxSupply = value;

        emit MaxSupplyUpdated(value);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DSU - Duplicate Structs Usage

Criticality	Minor / Informative
Location	OniAffiliate.sol#L669,728
Status	Unresolved

Description

The contract contains two identical structs, `RefData` and `SourceRefData`, both containing the same fields, `saleAmount` and `rewardAmount`. These structs are used to track similar data, but their duplication introduces redundancy into the codebase. Specifically, both structs are incremented by the same amounts in the same manner, leading to unnecessary repetition. The presence of these duplicate structs not only increases the complexity of the contract but also poses a risk of inconsistencies or errors if one struct is updated while the other is not. Additionally, this redundancy can lead to increased gas costs due to the extra storage and processing required.

```
   struct RefData {
        uint256 saleAmount;
        uint256 rewardAmount;
    }

    struct SourceRefData {
        uint256 saleAmount;
        uint256 rewardAmount;
    }
...
RefData storage refData_ = refData[referrer];
if (refData_.saleAmount == 0) ++numReferrers;
refData_.saleAmount += saleAmount;
refData_.rewardAmount += refRewardAmount;

SourceRefData storage sourceRefData_ = sourceRefData[source];
sourceRefData_.saleAmount += saleAmount;
sourceRefData_.rewardAmount += refRewardAmount;
```

Recommendation

It is recommended to consolidate the two structs into a single struct to represent the shared data. By removing the redundant struct and utilizing one unified struct, the contract will be streamlined, making it easier to read and maintain. This change will also help prevent potential errors and inconsistencies, as there will be only one source of truth for the data in question. Moreover, reducing redundancy will decrease gas costs, as fewer storage slots will be used and the overall complexity of the contract will be reduced. Simplifying the contract in this way enhances both its efficiency and reliability.

MT - Mints Tokens

Criticality	Minor / Informative
Location	OniToken.sol#L632,640
Status	Unresolved

Description

The contract owner has the authority to mint tokens. The owner may take advantage of it by calling the `mint` functionS. As a result, the contract tokens will be highly inflated.

```
    function mint(address to, uint256 amount) external onlyOwner
{
            _mint(to, amount);
    }

    function mint(uint256 amount) external onlyOwner {
            _mint(_msgSender(), amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	OniAffiliate.sol#L759
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

Specifically, if the `refRewardPercent` plus the `treasuryAmount` is greater than the `DENOMINATOR` , the transaction will be reverted as the funds will not be sufficient.

```
uint256 treasuryAmount = (fundAmount * TREASURY_RATE) /
DENOMINATOR;
...
fundAmount -= (refRewardAmount + treasuryAmount);
payable(owner()).sendValue(fundAmount);
        ...
function updateRefRewardPercent(uint16 value) external
onlyOwner {
    _refRewardPercent = value;

    emit RefRewardPercentUpdated(value);
}
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

RNM - Redundant Non-Reentrant Modifier

Criticality	Minor / Informative
Location	OniNode.sol#L2420 OniIco.sol#L965
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract currently uses the `nonReentrant` modifier in the `recoverETH` and `withdraw` functions. However, it becomes evident that there are no scenarios within these functions where reentrancy attacks can occur. As a result, the `nonReentrant` modifier is redundant and can be safely removed.

```
function recoverETH() external onlyOwner nonReentrant {
    ...;
}

function withdraw(
    address token,
    uint256 amount
) external onlyOwner nonReentrant {
    ...
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

RRL - Redundant Reward Logic

Criticality	Minor / Informative
Location	OniIco.sol#L870 OniNode.sol#L2157 OniAffiliate.sol#L719
Status	Unresolved

Description

The contract is designed to manage referral rewards across the `OniIico`, `OniNode`, and `OniAffiliate` contracts. Both the `OniIico` and `OniNode` contracts include logic to check that the referrer address is not the caller and to calculate the `refRewardPercent`. Additionally, the `OniAffiliate` contract replicates this functionality with extra checks to prevent the same account from being used as a referrer and to calculate the `refRewardAmount`. This redundancy results in additional and repetitive calculations, causing inefficiencies in the system.

```
if (referrer != address(0)) {
    if (referrer == caller) referrer = address(0);
    else {
        // send referral reward to the referrer
        IOniAffiliate oniAffiliate_ =
IOniAffiliate(_oniAffiliate);
        uint16 refRewardPercent =
oniAffiliate_.refRewardPercent();
        uint256 refRewardAmount = (payAmount *
refRewardPercent) /
            DENOMINATOR;
        oniAffiliate_.saleByReferrer{value: refRewardAmount}(
            caller,
            referrer,
            payAmount
        );
    }
```

```
if (referrer != address(0)) {
    if (referrer == caller) referrer = address(0);
    else {
        // send referral reward to the referrer
        IOniAffiliate oniAffiliate_ =
IOniAffiliate(_oniAffiliate);
        uint16 refRewardPercent =
oniAffiliate_.refRewardPercent();
        refRewardAmount =
            (fundsRequired * refRewardPercent) /
            DENOMINATOR;
        oniAffiliate_.saleByReferrer{value: refRewardAmount}(
            caller,
            referrer,
            fundsRequired
        );
    }
```

```
if (account == referrer) revert("invalid referrer");

uint256 refRewardAmount = (saleAmount * _refRewardPercent) /
            DENOMINATOR;
```

Recommendation

It is recommended to implement the referrer reward checks and calculations exclusively within the `OniAffiliate` contract. This approach will streamline the logic, reducing redundancy and improving efficiency. By consolidating the referral reward logic into a single contract, future maintenance will be simplified, and the overall system performance will be enhanced.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	OniIco.sol#L909
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
IERC20(oniToken).safeTransfer(caller, userRequiredAmount);
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	OniNode.sol#L178,1850 OniIco.sol#L290 OniAffiliate.sol#L178
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.

```
function DOMAIN_SEPARATOR() external view returns (bytes32);

function __unsafe_increaseBalance(address account, uint256
amount) internal {
        _balances[account] += amount;
    }
}
```

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	OniToken.sol#L506 OniNode.sol#L354,383,410,420,435,445,484,545,556,571,580,593,606,645,931,938,946,957,967,1051,1064,1100,1111,1153,1202,1215,1245,1268,1275,1283,1292,1344,1351,1360,1375,1382,1492,1686,1850,2123,2396 OniIco.sol#L384,413,440,450,465,475,514,575,586,601,610,623,636,675,762 OniAffiliate.sol#L354,383,410,420,435,445,484,545,556,571,580,593,606,645
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 _burn(address account, uint256 amount) internal
virtual {
        require(account != address(0), "ERC20: burn from the
zero address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account];
...
            _totalSupply -= amount;
        }

        emit Transfer(account, address(0), amount);

        _afterTokenTransfer(account, address(0), amount);
    }

...
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	OniNode.sol#L1013,1016,1028,1032,1033,1034,1035,1036,1037,1043
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.

```
denominator := div(denominator, twos)
inverse *= 2 - denominator * inverse
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	OniNode.sol#L2220,2242,2283
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 i
uint256 refRewardAmount;
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	OniNode.sol#L2201,2363 OniIco.sol#L846,848,936
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.

```
Affiliate = oniAffiliate_;
Affiliate = affiliate;
oniToken = oniToken_
_oniAffiliate = oniAffiliate_
_oniAffiliate = affiliate
```

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	OniNode.sol#L501,974,1325,1800 OniIco.sol#L531 OniAffiliate.sol#L501
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 {
                let returndata_size := mload(returndata)
                revert(add(32, returndata), returndata_size)
            }

assembly {
...
                prod1 := sub(sub(mm, prod0), lt(mm, prod0))
            }

assembly {
                ptr := add(buffer, add(32, length))
            }


...
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	OniToken.sol#L10,38,123,205,235,601 OniNode.sol#L10,38,123,187,269,517,662,691,825,854,885,916,1259,1306,1393,1861,1892,2053,2133,2150 OniIco.sol#L10,38,123,205,235,299,547,692,772,787 OniAffiliate.sol#L10,38,123,187,269,517,659
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.4;

...
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	OniToken.sol#L10,38,123,205,235,601 OniNode.sol#L10,38,123,187,269,517,662,691,825,854,885,916,1259,1306,1393,1861,1892,2053,2133,2150 OniIco.sol#L10,38,123,205,235,299,547,692,772,787 OniAffiliate.sol#L10,38,123,187,269,517,659
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;
pragma solidity ^0.8.4;
pragma solidity ^0.8.1;
...
```

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.

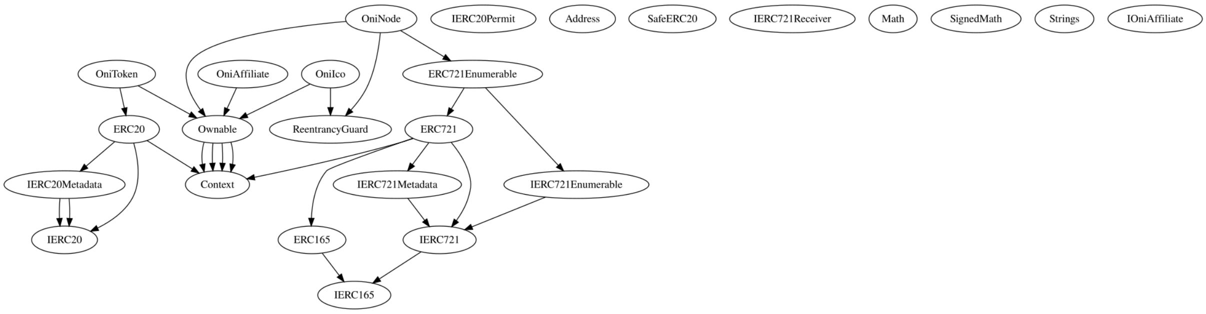
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
OniToken	Implementation	ERC20, Ownable		
		Public	✓	ERC20
	decimals	Public		-
	mint	External	✓	onlyOwner
	mint	External	✓	onlyOwner
OniNode	Implementation	Ownable, ERC721Enumerable, ReentrancyGuard		
		Public	✓	ERC721
	mint	External	Payable	nonReentrant
	requiredFundsForMint	External		-
	updateMintPrice	External	✓	onlyOwner
	mintPrice	External		-
	updatePriceChangeConf	External	✓	onlyOwner
	priceChangeConf	External		-
	updateMaxSupply	External	✓	onlyOwner
	maxSupply	External		-
	updateAffiliate	External	✓	onlyOwner

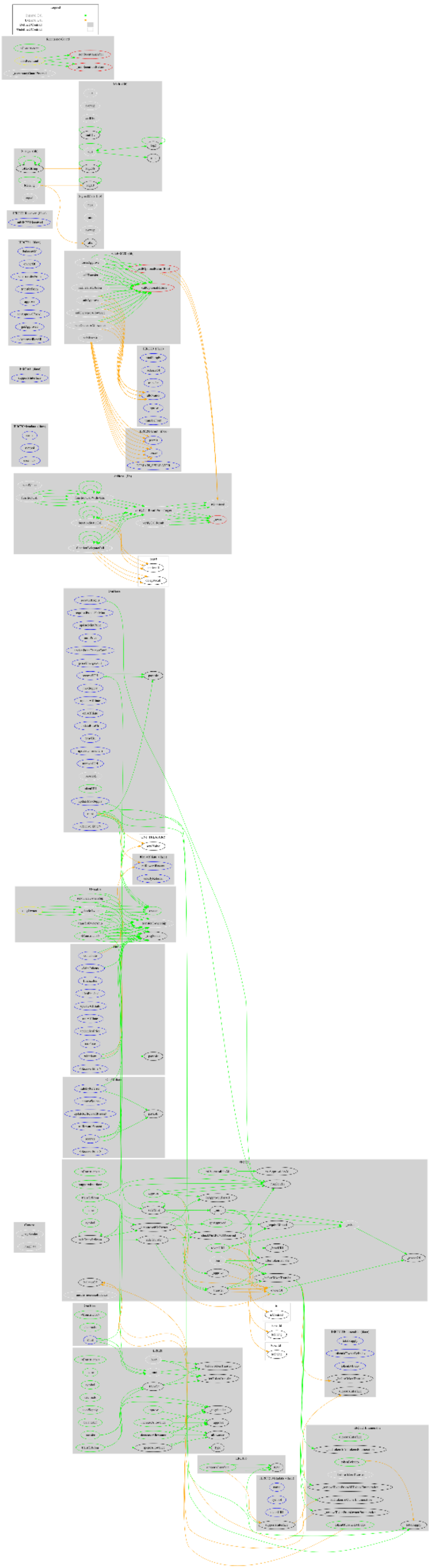
	oniAffiliate	External		-
	updateBaseUri	External	✓	onlyOwner
	baseUri	External		-
	updateUnrevealUri	External	✓	onlyOwner
	unrevealUri	External		-
	_baseURI	Internal		
	tokenURI	Public		-
	recoverETH	External	✓	onlyOwner nonReentrant
	recoverTokens	External	✓	onlyOwner
		External	Payable	-
OniIco	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	contribute	External	Payable	nonReentrant
	claimTokens	External	✓	-
	finalizeIco	External	✓	onlyOwner
	isIcoFinished	External		-
	updateAffiliate	External	✓	onlyOwner
	oniAffiliate	External		-
	updateIcoPrice	External	✓	onlyOwner
	icoPrice	External		-
	withdraw	External	✓	onlyOwner nonReentrant

		External	Payable	-
OniAffiliate	Implementation	Ownable		
	saleByReferrer	External	Payable	-
	approveSource	External	✓	onlyOwner
	updateRefRewardPercent	External	✓	onlyOwner
	refRewardPercent	External		-
	recover	External	✓	onlyOwner
		External	Payable	-

Inheritance Graph



Flow Graph



Summary

The Oni ecosystem comprises a suite of smart contracts designed to support a comprehensive blockchain-based platform. The OniToken contract implements a custom ERC20 token mechanism. The OniIco contract manages the Initial Coin Offering process. The OniNode contract facilitates dynamic NFT minting and management, integrating a referral system. The OniAffiliate contract oversees the referral reward system. This audit investigates security issues, business logic concerns, and potential improvements across these contracts to ensure a secure and efficient environment for users.

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