Cyberscope

Audit Report

Guitarist

November 2023

Network	ETH
Address	0xa249d8f280ba32ab3e95bddf4f3dbde059d62ac3
Audited by	© cyberscope

Table of Contents

Review

Contract Name	Guitarist
Compiler Version	v0.8.17+commit.8df45f5f
Optimization	200 runs
Explorer	https://etherscan.io/address/0xa249d8f280ba32ab3e95bddf4f3dbde059d62ac3
Address	0xa249d8f280ba32ab3e95bddf4f3dbde059d62ac3
Network	ETH
Symbol	Guitarist

Audit Updates

Initial Audit	12 Nov 2023

Source Files

Filename	SHA256
Guitarist.sol	82ad58d3ccbfd06a6940e7102a9ca4ef0acc6eb1a66f4fcb2f3b0568e05b3eb5

Overview

The "Guitarist" NFT contract, built on the Ethereum blockchain, exemplifies a comprehensive implementation of a non-fungible token (NFT) project. It integrates the ERC721A implementation for efficient batch minting and ERC2981 for standardized royalty management. The contract is designed to handle a maximum supply of 10,000 unique digital assets, with provisions for public sales and special distributions for the contract owner like airdrops and mint nfts to specific addresses. The contract maintains robust security features, including operator approval filters for sensitive actions like transfers.

Owner Functionalities

The owner of the "Guitarist" NFT contract wields significant control over its operations. Key functionalities include the ability to mint tokens to specific addresses, enabling or disabling the reveal of NFTs, and pausing the minting process. The owner can also set and update the URIs for both revealed and unrevealed NFT metadata, which is crucial for managing how the NFTs are presented to the public. Furthermore, the owner has the authority to adjust the mint price and the maximum amount of NFTs purchasable per wallet, allowing them to manage the economics of the NFT collection effectively.

Mint Functionality

For users, the primary interaction with the "Guitarist" contract is through the mint function. This function allows users to purchase a specified quantity of NFTs, and pay the corresponding amount of Ether set as the public sale cost. The minting process is straightforward: users specify how many NFTs they wish to buy and send the required amount of Ether. Upon successful transaction, the NFTs are minted and allocated to the user's wallet, and the paid Ether is transferred directly to the owner's treasury wallet. This process not only ensures a smooth acquisition of NFTs for users but also maintains a direct and transparent flow of funds to the contract owner.

Roles

Owner

The owner can interact with the following functions in the "Guitarist" NFT contract:

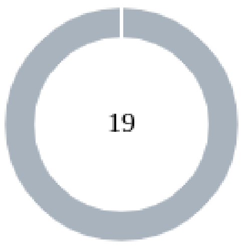
- function airdrop(address[] calldata receiver, uint256[] calldata quantity)
- function partnerMint(address partner, uint256 quantity)
- function toggleReveal()
- function toggle_public_mint_status()
- function setNotRevealedURI(string memory _notRevealedURI)
- function setContractURI(string memory _contractURI)
- function setPublicSaleCost(uint256 _publicSaleCost)
- function setMax_per_wallet(uint256 _max_per_wallet)
- function setTreasuryWallet(address _treasuryWallet)
- function setWallet2(address _wallet2)
- function setWallet3(address _wallet3)
- function getBaseURI()

User

Users can interact with the following functions in the "Guitarist" NFT contract:

- function mint(uint256 quantity)
- function transferFrom(address from, address to, uint256 tokenId)

Findings Breakdown



🔴	Critical	0
🟠	Medium	0
●	Minor / Informative	19

Severity		Unresolved	Acknowledged	Resolved	Other
🔴	Critical	0	0	0	0
🟠	Medium	0	0	0	0
●	Minor / Informative	19	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	PRE	Potential Reentrance Exploit	Unresolved
●	MSB	Max Supply Bypass	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DFI	Duplicated Function Implementation	Unresolved
●	MC	Missing Check	Unresolved
●	RF	Redundant Functions	Unresolved
●	FO	Function Optimization	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	RPF	Redundant Payable Function	Unresolved
●	EIS	Excessively Integer Size	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

●	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

PRE - Potential Reentrance Exploit

Criticality	Minor / Informative
Location	Guitarist.sol#L1611
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the payable transfer method. The recipient could be a malicious contract that has an untrusted code in its fallback function that makes a recursive call back to the original contract. The re-entrance exploit could be used by a malicious user to drain the contract's funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

Specifically, when the `_safeMint` function, is called within the `mint` function, it executes the `_checkOnERC721Received` function. The `_checkOnERC721Received` function is designed to verify if the recipient address (in this case, the sender of the mint transaction) is a contract that properly implements the IERC721Receiver interface. However, if this recipient is a malicious contract, it could override the onERC721Received function to initiate a new call to the contract's mint function.

Given the current implementation, this recursive call could potentially bypass the `MAX_SUPPLY` check, as the state variable representing the total supply of minted NFTs (`totalSupply`) is updated only after the external call. As a result, since the reentrant call occurs before the `_currentIndex` variable is increased (which reflects the `totalSupply` value), it could allow minting more NFTs than the intended maximum supply, violating the contract's supply constraints and potentially leading to other unintended consequences.

```
  function mint(uint256 quantity) public payable  {
        require(totalSupply() + quantity <= MAX_SUPPLY,"No More NFTs to
Mint");
        ....
        _safeMint(msg.sender, quantity);
        withdraw();
    }

  function _safeMint(
        address to,
        uint256 quantity,
        bytes memory _data
    ) internal {
        _mint(to, quantity, _data, true);
    }

    function _mint(
        address to,
        uint256 quantity,
        bytes memory _data,
        bool safe
    ) internal {
        ...
        unchecked {
            _addressData[to].balance += uint64(quantity);
            _addressData[to].numberMinted += uint64(quantity);

            _ownerships[startTokenId].addr = to;
            _ownerships[startTokenId].startTimestamp =
uint64(block.timestamp);

            uint256 updatedIndex = startTokenId;

            for (uint256 i; i < quantity; i++) {
                emit Transfer(address(0), to, updatedIndex);
                if (safe && !_checkOnERC721Received(address(0), to,
updatedIndex, _data)) {
                    revert TransferToNonERC721ReceiverImplementer();
                }
                updatedIndex++;
            }
              _currentIndex = uint128(updatedIndex);
    ...}
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Do Not allow contract addresses to receive funds.
- Proceed with the external call as the last statement of the method, so that the state will have been updated properly during the re-entrance phase.

MSB - Max Supply Bypass

Criticality	Minor / Informative
Location	Guitarist.sol#L1585
Status	Unresolved

Description

The contract allows the owner to potentially exceed the predefined `MAX_SUPPLY` limit of tokens through the `airdrop` function. preMintNFTsThis function lacks a check to ensure that the total supply doesn't surpass `MAX_SUPPLY` when tokens are being minted during the airdrop. The absence of such a check in the loop that performs the minting operation can lead to scenarios where the total supply of tokens exceeds the intended limit.

```
    function airdrop(address[] calldata receiver, uint256[] calldata 
quantity) public payable onlyOwner {

        require(receiver.length == quantity.length, "Airdrop data does 
not match");

        for(uint256 x = 0; x < receiver.length; x++){
        _safeMint(receiver[x], quantity[x]);
        }
    }
```

Recommendation

It is recommended to introduce additional safeguards within the `airdrop` function to prevent minting that exceeds the `MAX_SUPPLY` limit. This can be achieved by adding a check before minting in the airdrop loop to ensure that the total minted quantity will not surpass `MAX_SUPPLY`.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Guitarist.sol#L1585,1612,1676,1707
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 owner has exclusive rights to mint tokens to specific addresses, enable or disable NFT reveal, pause the minting process, set URIs for both revealed and unrevealed NFTs, and adjust the mint price and maximum amount per wallet. This concentration of power in a single authority presents a centralization risk, potentially undermining the decentralized nature expected in blockchain applications.

Additionally, the `onlyAllowedOperator` and `onlyAllowedOperatorApproval` modifiers, through their reliance on the `OPERATOR_FILTER_REGISTRY`, introduce centralization risks related to control, security, and the efficient operation of the smart contract.

```
   function airdrop(address[] calldata receiver, uint256[] calldata
quantity) public payable onlyOwner {

        require(receiver.length == quantity.length, "Airdrop data does
not match");

        for(uint256 x = 0; x < receiver.length; x++){
        _safeMint(receiver[x], quantity[x]);
        }
    }

    function partnerMint(address partner, uint256 quantity) public
payable onlyOwner  {
        require(totalSupply() + quantity <= MAX_SUPPLY,"No More NFTs to
Mint");

        _safeMint(partner, quantity);
    }

    function toggleReveal() external onlyOwner {

        if(revealed==false){
            revealed = true;
        }else{
            revealed = false;
        }
    }

    function toggle_public_mint_status() external onlyOwner {

        if(public_mint_status==false){
            public_mint_status = true;
        }else{
            public_mint_status = false;
        }
    }

    function setNotRevealedURI(string memory _notRevealedURI) public
onlyOwner {
        notRevealedUri = _notRevealedURI;
    }

    function setContractURI(string memory _contractURI) external
onlyOwner {
        contractURI = _contractURI;
    }
    function setPublicSaleCost(uint256 _publicSaleCost) external
onlyOwner {
        publicSaleCost = _publicSaleCost;
    }

    function setMax_per_wallet(uint256 _max_per_wallet) external
onlyOwner {
        max_per_wallet = _max_per_wallet;
    }
```

```
    modifier onlyAllowedOperator(address from) virtual {
        // Allow spending tokens from addresses with balance
        // Note that this still allows listings and marketplaces with
escrow to transfer tokens if transferred
        // from an EOA.
        if (from != msg.sender) {
            _checkFilterOperator(msg.sender);
        }
        _;
    }

    modifier onlyAllowedOperatorApproval(address operator) virtual {
        _checkFilterOperator(operator);
        _;
    }

    function _checkFilterOperator(address operator) internal view
virtual {
        // Check registry code length to facilitate testing in
environments without a deployed registry.
        if (address(OPERATOR_FILTER_REGISTRY).code.length > 0) {
            if
(!OPERATOR_FILTER_REGISTRY.isOperatorAllowed(address(this), operator)) {
                revert OperatorNotAllowed(operator);
            }
        }
    }
```

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.

DFI - Duplicated Function Implementation

Criticality	Minor / Informative
Location	Guitarist.sol#L1585,1612
Status	Unresolved

Description

The contract contains two separate functions, `airdrop` and `partnerMint`, both of which essentially serve the same purpose which is to allow the owner to mint NFTs for specific address. The `airdrop` function is designed to mint NFTs to multiple addresses, while `partnerMint` is focused on a single address. Given their similar functionality, maintaining these as separate functions leads to redundancy in the contract's code, potentially affecting efficiency and gas costs.

```
   function airdrop(address[] calldata receiver, uint256[] calldata
quantity) public payable onlyOwner {

        require(receiver.length == quantity.length, "Airdrop data does
not match");

        for(uint256 x = 0; x < receiver.length; x++){
        _safeMint(receiver[x], quantity[x]);
        }
    }

    function partnerMint(address partner, uint256 quantity) public
payable onlyOwner  {
        require(totalSupply() + quantity <= MAX_SUPPLY,"No More NFTs to
Mint");

        _safeMint(partner, quantity);
    }
```

Recommendation

It is recommended to reconsider the code implementation and merge the `airdrop` and `partnerMint` functions into a single, more versatile function. This consolidated function should be capable of handling both single and multiple mints, thereby streamlining the contract's logic and potentially optimizing gas usage. A unified function could accept an array of addresses and quantities, with logic to handle both single and multiple mint scenarios. This approach not only simplifies the contract's interface but also enhances its maintainability and efficiency, eliminating redundant code paths and reducing the contract's overall complexity.

MC - Missing Check

Criticality	Minor / Informative
Location	Guitarist.sol#L1711
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, the `setMax_per_wallet` function permits the owner to set the `max_per_wallet` limit to a value higher than `MAX_SUPPLY`, which could lead to inconsistencies and unintended behavior in the token distribution process.

```
    function setMax_per_wallet(uint256 _max_per_wallet) external
onlyOwner {
            max_per_wallet = _max_per_wallet;
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is advised to modify the `setMax_per_wallet` function to include a validation check that prevents setting the `max_per_wallet` limit above the `MAX_SUPPLY`. These measures will ensure adherence to the intended supply constraints, maintaining the integrity and rarity of the tokens as originally designed.

RF - Redundant Functions

Criticality	Minor / Informative
Location	Guitarist.sol#L1719,1723
Status	Unresolved

Description

The contract contains the `setWallet2` and `setWallet3` functions which update the the `wallet2` and `wallet3` address variables. However, these wallet variables are only utilized within the constructor and not in any other part of the contract's logic. Consequently, the ability to update these addresses via these functions is unnecessary, as the updated values are not utilized after the deployment of the contract.

```
   function setWallet2(address _wallet2) public onlyOwner {
        wallet2 = _wallet2;
    }

    function setWallet3(address _wallet3) public onlyOwner {
        wallet3 = _wallet3;
    }
```

Recommendation

It is recommended to remove the `setWallet2` and `setWallet3` functions from the contract since the updates performed by these functions to `wallet2` and `wallet3` addresses are effectively redundant. Since these wallet addresses are not employed in the contract beyond its initialization phase, maintaining functions to update them serves no practical purpose and unnecessarily bloats the contract's codebase. Removing these functions will streamline the contract, enhancing its clarity and efficiency.

FO - Function Optimization

Criticality	Minor / Informative
Location	Guitarist.sol#L1676,1685
Status	Unresolved

Description

The current implementation of both the `toggleReveal` and `toggle_public_mint_status` functions in the contract utilizes an `if-else` statement to toggle the state of the `revealed` and `public_mint_status` variables respectively. However, this approach, while functional, is more verbose than necessary. Each function checks a boolean condition and then explicitly sets the variable to true or false based on this condition. This results in additional lines of code that essentially replicate the standard behavior of a boolean toggle.

```
   function toggleReveal() external onlyOwner {

        if(revealed==false){
            revealed = true;
        }else{
            revealed = false;
        }
    }

    function toggle_public_mint_status() external onlyOwner {

        if(public_mint_status==false){
            public_mint_status = true;
        }else{
            public_mint_status = false;
        }
    }
```

Recommendation

It is recommended to refactor both functions for improved conciseness and clarity. The `toggleReveal` function can be simplified to a single line: `revealed = !revealed;` . Similarly, the `toggle_public_mint_status` function can be refactored to `public_mint_status = !public_mint_status;` . These changes leverages the logical NOT operator (!) to flip the boolean state of the variable, thereby achieving the same functionality in a more streamlined and readable manner. This refactoring not only enhances code readability but also marginally optimizes the contract by reducing the number of operations required.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Guitarist.sol#L1685,1707,1713
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.

```
   function toggle_public_mint_status() external onlyOwner {

        if(public_mint_status==false){
            public_mint_status = true;
        }else{
            public_mint_status = false;
        }
    }

    function setPublicSaleCost(uint256 _publicSaleCost) external
onlyOwner {
        publicSaleCost = _publicSaleCost;
    }

    function setMax_per_wallet(uint256 _max_per_wallet) external
onlyOwner {
        max_per_wallet = _max_per_wallet;
    }
```

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.

RPF - Redundant Payable Function

Criticality	Minor / Informative
Location	Guitarist.sol#L1585,1612
Status	Unresolved

Description

The contract's function `airdrop` and `partnerMint` have the `payable` modifier which means that the contract owner pays with the native token. The function does not use the `msg.value` variable anywhere inside the function to indicate the usage of the payable modifier. As a result, the payable modifier is redundant.

```
    function airdrop(address[] calldata receiver, uint256[] calldata
quantity) public payable onlyOwner {

        require(receiver.length == quantity.length, "Airdrop data
does not match");

        for(uint256 x = 0; x < receiver.length; x++){
        _safeMint(receiver[x], quantity[x]);
        }
    }

    function partnerMint(address partner, uint256 quantity) public
payable onlyOwner  {
        require(totalSupply() + quantity <= MAX_SUPPLY,"No More NFTs
to Mint");

        _safeMint(partner, quantity);
    }
```

Recommendation

The team is advised to remove the payable modifier from the function's declaration as it is not needed.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	Guitarist.sol#L1560,1562
Status	Unresolved

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

```
uint256 public MAX_SUPPLY = 10000;
...
uint256 public max_per_wallet = 10000;
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Guitarist.sol#L1560
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 MAX_SUPPLY = 10000
```

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	Guitarist.sol#L659,989,992,1002,1261,1560,1562,1569,1669,1685,1694,1698,1707,1711,1715,1719,1723
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.

```
uint256 _tokenId
uint256 _salePrice
uint128 internal _currentIndex
uint128 internal _burnCounter
mapping(uint256 => TokenOwnership) internal _ownerships
bytes memory _data
uint256 public MAX_SUPPLY = 10000
uint256 public max_per_wallet = 10000
bool public public_mint_status = true
uint96 _royaltyFeesInBips
address _receiver

...
```

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	Guitarist.sol#L130,371,397,407,422,432,447,457,471,481,489,698,710,724,1111,1116,1188,1416
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 verify(bytes32[] calldata proof, bytes32 leaf, bytes32 root)
internal pure returns (bool) {
        bytes32 computedHash = leaf;

        for (uint256 i = 0; i < proof.length; i++) {
            bytes32 proofElement = proof[i];

...
 computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
            }
        }

 // Check if the computed hash (root) is equal to the provided root
        return computedHash == root;
    }

...
```

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	Guitarist.sol#L1621,1678,1687
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.

```
revealed == false
revealed==false
public_mint_status==false
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	Guitarist.sol#L1039,1044,1065,1066,1071,1336
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 tokenIdsIdx
uint256 i
address currOwnershipAddr
```

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	Guitarist.sol#L1716,1720,1724
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.

```
treasuryWallet = _treasuryWallet
wallet2 = _wallet2
wallet3 = _wallet3
```

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	Guitarist.sol#L351,498,1495
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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	Guitarist.sol#L5,37,99,118,173,210,238,321,513,538,565,591,621,732,864,893,923,1551
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 ^0.8.13;
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	Guitarist.sol#L5,37,99,118,173,210,238,321,513,538,565,591,621,732,864,893,923,1551
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 ^0.8.13;
pragma solidity ^0.8.0;
pragma solidity ^0.8.4;
```

Recommendation

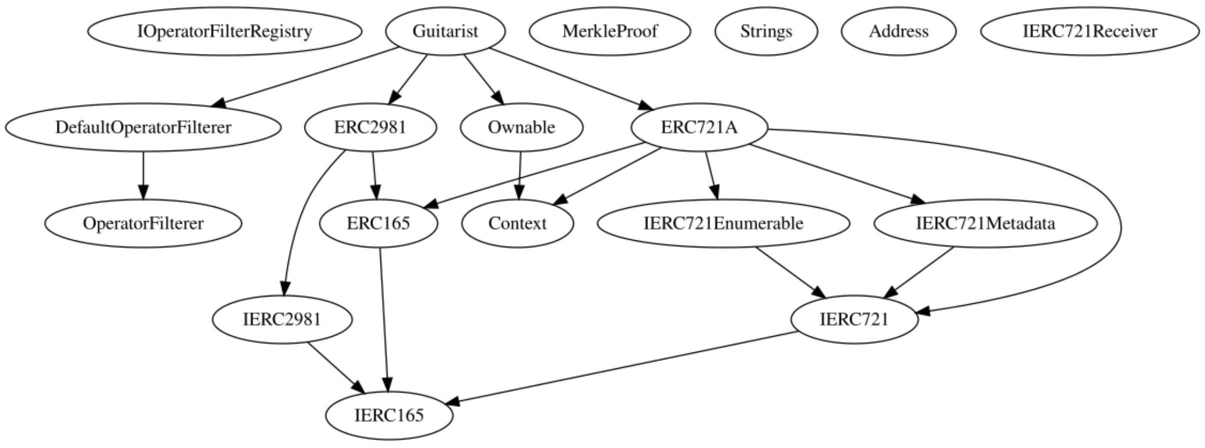
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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

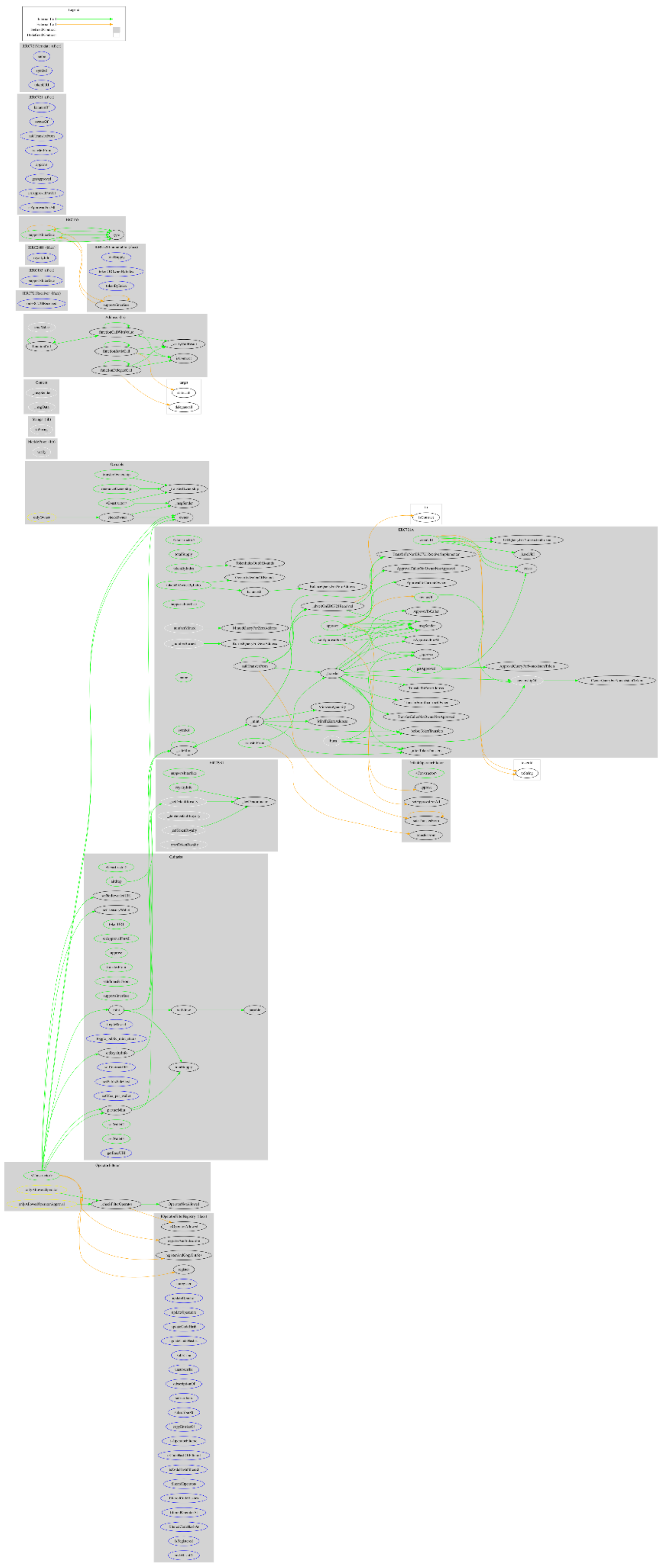
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Guitarist	Implementation	ERC721A, Ownable, ERC2981, DefaultOperatorFilterer		
		Public	✓	ERC721A
	airdrop	Public	Payable	onlyOwner
	mint	Public	Payable	-
	partnerMint	Public	Payable	onlyOwner
	tokenURI	Public		-
	setApprovalForAll	Public	✓	onlyAllowedOperatorApproval
	approve	Public	✓	onlyAllowedOperatorApproval
	transferFrom	Public	✓	onlyAllowedOperator
	safeTransferFrom	Public	✓	onlyAllowedOperator
	safeTransferFrom	Public	✓	onlyAllowedOperator
	supportsInterface	Public		-
	setRoyaltyInfo	Public	✓	onlyOwner
	toggleReveal	External	✓	onlyOwner
	toggle_public_mint_status	External	✓	onlyOwner
	setNotRevealedURI	Public	✓	onlyOwner
	setContractURI	External	✓	onlyOwner

	withdraw	Internal	✓	
	setPublicSaleCost	External	✓	onlyOwner
	setMax_per_wallet	External	✓	onlyOwner
	setTreasuryWallet	Public	✓	onlyOwner
	setWallet2	Public	✓	onlyOwner
	setWallet3	Public	✓	onlyOwner
	getBaseURI	External		onlyOwner

Inheritance Graph



Flow Graph



Summary

Guitarist contract implements a nft mechanism. This audit investigates security issues, business logic concerns and potential improvements. The audit revealed no critical or medium security findings in the contract and only minor / informative findings are reported. The images and metadata associated with the NFTs are unchangeable, ensuring their immutability.

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