Cyberscope

A TAC Security Company

Audit Report

Loomia

July 2025

Repository	https://github.com/oxydetoxy/launchpad-contract
Commit	2161aeebae4c3f53d6639827eaf0658a63c69cc6
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/oxydetoxy/launchpad-contract
Commit	2161aeebae4c3f53d6639827eaf0658a63c69cc6

Audit Updates

Initial Audit	01 Jul 2025

Source Files

Filename	SHA256
audit.sol	1fad89868c2c01317825bd5e1a4b11d4afd75fa19f7de1c3b67fbd08b54c2d7e

Overview

BambiPods Contract

The `BambiPods` contract is an ERC721 implementation tailored for NFT projects with structured minting processes. Built using OpenZeppelin libraries, it introduces a system of multi-phase minting, optional whitelisting through Merkle proofs, and controls for supply limits, royalties, and fund management.

Minting Phase Functionality

At the core of the contract is a phase-based minting mechanism. Each minting phase is defined by a start and end time, a specific mint price, a cap on the number of NFTs that can be minted during the phase, a per-wallet minting limit, and an optional Merkle root for whitelisting. The `mint` function enforces these constraints and ensures that users meet all conditions before minting. If whitelisting is active for a phase, users must provide a valid Merkle proof along with their allowed mint quantity. The contract tracks how many NFTs each address has minted per phase to prevent abuse. Additionally, a special function called `reallocatePendingSupplyToLastPhase` allows the contract owner to reclaim any unminted supply from earlier phases and assign it to the final phase, provided that the final phase has not yet started.

Supply and Access Control

Supply is managed both globally and per phase. The `maxMintableSupply` defines the total number of NFTs that can ever be minted by the contract, while each phase has its own `mintableSupply` to limit tokens available in that specific window. Several modifiers are used to enforce correct minting behavior.

The contract disables transfers entirely by overriding `transferFrom` and `_update`, making all NFTs minted through it non-transferable unless burned. The contract also leverages OpenZeppelin's `Ownable` module to restrict administrative functions, `Pausable` to allow emergency stops to minting activity, and `ReentrancyGuard` to ensure safe and secure handling of state-altering functions.

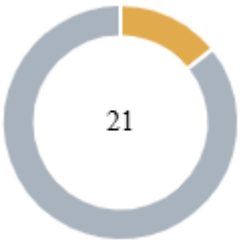
Royalty and URI Management

The contract supports royalty configuration through the `royaltyReceiver` and `royaltyPercentage` variables, allowing the owner to define a royalty destination and set a fee in basis points. Metadata resolution is handled via a configurable `baseURI`. The `tokenURI` function returns the full path to the metadata file by appending the token ID and `.json` extension to the base URI.

Withdrawal and Fund Handling

All ETH received from minting is tracked using the `totalFunds` variable. The contract owner can withdraw the accumulated minting proceeds to a predefined `fundsReceiver` address using the `withdraw` function. If needed, a secondary method called `normalWithdraw` allows the owner to withdraw the full contract balance to their own address. Both functions use the `nonReentrant` modifier and check for successful transfers, reverting the transaction if the transfer fails. This ensures that all fund-handling operations are reliable and secure.

Findings Breakdown



Critical	0
Medium	3
Minor / Informative	18

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	3	0	0	0
Minor / Informative	18	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	MT	Mints Tokens	Unresolved
Medium	PCI	Phase Configuration Inconsistencies	Unresolved
Medium	PRI	Phase Reallocation Inconsistencies	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	HV	Hardcoded Values	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	MPC	Merkle Proof Centralization	Unresolved
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	MWC	Missing Withdrawal Check	Unresolved
Minor / Informative	PF	Pausable Functionality	Unresolved
Minor / Informative	RML	Reusable Merkle Leaf	Unresolved
Minor / Informative	ST	Stops Transactions	Unresolved

●	TUUC	Token URI Update Concern	Unresolved
●	UEV	Unrefunded Excess Value	Unresolved
●	URSV	Unused Royalty State Variables	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L19	Stable Compiler Version	Unresolved

MT - Mints Tokens

Criticality	Medium
Location	audit.sol#L164,174,185,195
Status	Unresolved

Description

The contract allows users to mint tokens. During the contract's initialization if `maxMintableSupply` is set to 0 there will be no limitations on the amount of tokens that can be minted.

```
modifier hasSupply(uint256 phaseIndex, uint256 quantity) {
    //...
    if (maxMintableSupply > 0 && totalSupply() + quantity >
maxMintableSupply) {
        revert SupplyNotAvailable();
    }
    _;
}
```

Additionally the phases' configuration may allow users to mint token without limitations other than the total amount that can be minted:

Specifically, if `mintableSupply` of a phase is set to zero then the following check is bypassed.

```
modifier hasSupply(uint256 phaseIndex, uint256 quantity) {
    //...
    if (
        phase.mintableSupply > 0 &&
        phaseMintedSupply[phaseIndex] + quantity > phase.mintableSupply
    ) {
        revert SupplyNotAvailable();
    }
    //...
}
```

In the `mint` function, if `phase.merkleRoot` is equal to zero bytes and `phase.maxMintPerWallet` is not greater than zero, the user is allowed to mint any number of tokens, as long as the total minted amount does not exceed the `maxMintableSupply` .

```
if (phase.merkleRoot != bytes32(0)) {
    validateMerkleProof(merkleProof, phase.merkleRoot, allowedMints);

    if (
        allowedMints > 0 &&
        phaseWalletMintedCount[phaseIndex][msg.sender] + quantity >
        allowedMints
    ) {
        revert MintAllowanceExceeded();
    }
} else if (phase.maxMintPerWallet > 0 &&
phaseWalletMintedCount[phaseIndex][msg.sender] + quantity >
phase.maxMintPerWallet) {
    revert MintAllowanceExceeded();
}
```

The owner is also able to create a phase with configurations that allow them to mint tokens equal to `maxMintableSupply` . Additionally, if `maxMintableSupply` is set to 0, it removes any limitations, enabling the owner to mint an unlimited number of tokens.

Recommendation

The team should carefully manage the private keys of the owner's account who is able to change the contract's critical configurations. 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.

PCI - Phase Configuration Inconsistencies

Criticality	Medium
Location	audit.sol#L164
Status	Unresolved

Description

`SetMintPhases` has several inconsistencies that may harm the optimal operations of the contract.

Specifically:

- It does not sanitize the parameters added by the user as mentioned in the MC section.
- The function deletes the previous phases to create new ones, however users may already have minted tokens on each of these phases and this is saved in a mapping `phaseWalletMintedCount`. This can also be used to not allow users that already minted tokens to mint anymore or the opposite.
- The amount of phases can also be changed which may create an inconsistency in case users have already minted tokens on a phase that is deleted instead of updated.
- Changing the phases configuration can alter the way users mint tokens during the use of the `mint` function from using merkleProofs to checking the `maxMintPerWallet` or even minting an unlimited amount.
- The caller of the function is able to end a phase that is currently active, not allow the phase to ever start, reactivate a phase or set the phase in a future time even if the phase has already started. Also multiple phases can be active at the same time.
- `maxMintableSupply` may not be equal to the combined amounts of tokens allowed to be minted in the phases.

```
function setMintPhases(MintPhase[] calldata newPhases) external
onlyOwner {
    delete mintPhases;
    for (uint256 i = 0; i < newPhases.length; i++) {
        if (newPhases[i].startTime > newPhases[i].endTime) {
            revert InvalidPhase();
        }
        mintPhases.push(newPhases[i]);
    }
}
```

Recommendation

The team should consider allowing the setting of phases to happen only once and before enabling minting. Additionally the parameters should be correctly sanitized in order to ensure that the contract operates smoothly and as intended.

PRI - Phase Reallocation Inconsistencies

Criticality	Medium
Location	audit.sol#L226
Status	Unresolved

Description

`reallocatePendingSupplyToLastPhase` function exhibits several inconsistencies.

Specifically:

- `reallocated` boolean is never updated.
- Reallocation can be followed by `setMintPhases` which will reset the changes to the state.
- The function does not check if the rest of the phases are completed before reallocating the supply to the last phase.
- The function does not set the phases as finished meaning that users can still call the `mint` function for these phases if they are active.
- Since multiple phases can be active at the same time, the last phase may be activated or finished before the rest of the phases.
- Being the last phase on the list of phases does not ensure that it will always be the last active phase.

```
function reallocatePendingSupplyToLastPhase() external onlyOwner {
    if(reallocated){
        revert AlreadyAllocated();
    }
    if(block.timestamp > mintPhases[mintPhases.length - 1].startTime){
        revert LastPhaseAlreadyStarted();
    }
    if(mintPhases.length <= 1){
        revert InvalidPhaseLength();
    }
    uint256 totalPending = 0;
    for (uint256 i = 0; i < mintPhases.length - 1; i++) {
        uint256 minted = phaseMintedSupply[i];
        uint256 supply = mintPhases[i].mintableSupply;

        if (supply > minted) {
            totalPending += (supply - minted);
        }
    }
    if(totalPending > 0){
        MintPhase storage lastPhase = mintPhases[mintPhases.length - 1];
        lastPhase.mintableSupply += totalPending;
    }
}
```

Recommendation

It is recommended that the team restructures the function and the codebase considering the pointers mentioned above.

CR - Code Repetition

Criticality	Minor / Informative
Location	audit.sol#L258,266
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.

```
function withdraw() public onlyOwner nonReentrant {...}
function normalWithdraw() public onlyOwner nonReentrant {...}
```

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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	audit.sol#L139,144,152,156,164,226,258,266
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.

```
function setDefaultRoyalty(address receiver, uint96 feeNumerator)
external onlyOwner
function setBaseURI(string memory _baseUri) external onlyOwner()
function pause() public onlyOwner
function unpause() public onlyOwner
function setMintPhases(MintPhase[] calldata newPhases) external
onlyOwner
function reallocatePendingSupplyToLastPhase() external onlyOwner
function withdraw() public onlyOwner nonReentrant
function normalWithdraw() public onlyOwner nonReentrant
```

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.

HV - Hardcoded Values

Criticality	Minor / Informative
Location	audit.sol#L103,117
Status	Unresolved

Description

The contract contains multiple instances where numeric values are directly hardcoded into the code logic rather than being assigned to constant variables with descriptive names. Hardcoding such values can lead to several issues, including reduced code readability, increased risk of errors during updates or maintenance, and difficulty in consistently managing values throughout the contract. Hardcoded values can obscure the intent behind the numbers, making it challenging for developers to modify or for users to understand the contract effectively.

```
percentage > 10000
quantity > 15
```

Recommendation

It is recommended to replace hardcoded numeric values with variables that have meaningful names. This practice improves code readability and maintainability by clearly indicating the purpose of each value, reducing the likelihood of errors during future modifications. Additionally, consider using constant variables which provide a reliable way to centralize and manage values, improving gas optimization throughout the contract.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	audit.sol#L135,136
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.

```
fundsReceiver
maxMintableSupply
```

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.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	audit.sol#L164,214
Status	Unresolved

Description

The contract uses a Merkle Proof mechanism in order to define many applicable addresses. The verification process is based on an off-chain configuration. The contract owner is responsible for updating the in-chain “Merkle Root” in order to validate correctly the provided message.

```
function setMintPhases(MintPhase[] calldata newPhases) external
onlyOwner {
    delete mintPhases;
    for (uint256 i = 0; i < newPhases.length; i++) {
        if (newPhases[i].startTime > newPhases[i].endTime) {
            revert InvalidPhase();
        }
        mintPhases.push(newPhases[i]);
    }
}
function validateMerkleProof(
    bytes32[] calldata merkleProof,
    bytes32 merkleRoot,
    uint256 allowedMints
) internal view {
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender,
allowedMints));
    bool valid = MerkleProof.verify(merkleProof, merkleRoot, leaf);
    if (!valid) {
        revert InvalidProof();
    }
}
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly. 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.

MC - Missing Check

Criticality	Minor / Informative
Location	audit.sol#L123,144,164
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.

In the constructor `_maxMintableSupply` is not checked for zero value. `_baseUri` should also be checked to ensure that it is not empty. The case is similar for `setBaseUri` .

```
constructor(string memory name, string memory symbol, string memory
_baseUri, address _royalityReceiver, uint96 _royalityPercent, address
_fundsReceiver, uint256 _maxMintableSupply) ERC721(name, symbol)
Ownable(msg.sender) isValidRoyalty(_royalityReceiver, _royalityPercent){
    //...
    baseUri=_baseUri;
    maxMintableSupply=_maxMintableSupply;
}
function setBaseURI(string memory _baseUri) external onlyOwner(){
    baseUri=_baseUri;
}
```

`setMintPhases` allows the owner to add as input an array of `MintPhases`. However the struct elements are not sanitized. Specifically, `startTime` should be bigger than the current timestamp. Additionally, due to the lack of parameter validation multiple phases can be active at the same time. `mintPrice` should be a reasonable non-zero value. `mintableSupply` should also be a reasonable non-zero value. `maxMintPerWallet` should have a non-zero value if `merkleRoot` is empty and the opposite.

```
function setMintPhases(MintPhase[] calldata newPhases) external
onlyOwner {
    delete mintPhases;
    for (uint256 i = 0; i < newPhases.length; i++) {
        if (newPhases[i].startTime > newPhases[i].endTime) {
            revert InvalidPhase();
        }
        mintPhases.push(newPhases[i]);
    }
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	audit.sol#L139,144,164,226
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 setDefaultRoyalty(address receiver, uint96 feeNumerator)
external onlyOwner
function setBaseURI(string memory _baseUri) external onlyOwner()
function setMintPhases(MintPhase[] calldata newPhases) external
onlyOwner
function reallocatePendingSupplyToLastPhase() external onlyOwner
```

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.

MWC - Missing Withdrawal Check

Criticality	Minor / Informative
Location	audit.sol#L258
Status	Unresolved

Description

The contract has a `withdraw` function that sends `totalFunds` to the `fundsReciver` address. However, this function can be called multiple times, potentially allowing the funds to be withdrawn more than once. If the intent is to send the funds only once, the contract lacks a state variable to enforce this restriction and ensure that the function can only be called a single time.

```
function withdraw() public onlyOwner nonReentrant {
    (bool success, ) = fundsReceiver.call{value: totalFunds}("");
    if(!success){
        revert TransferFailed();
    }
    emit FundsWithdrawn(fundsReceiver);
}
```

Recommendation

The team should follow the recommendations mentioned above.

PF - Pausable Functionality

Criticality	Minor / Informative
Location	audit.sol#L152,174
Status	Unresolved

Description

The contract has functionality that allows the owner to pause critical methods of the contract.

```
function pause() public onlyOwner {
    _pause();
}
function mint(uint256 phaseIndex, bytes32[] calldata merkleProof,
uint256 allowedMints, uint256 quantity) external payable ...
whenNotPaused
```

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.

RML - Reusable Merkle Leaf

Criticality	Minor / Informative
Location	audit.sol#L219
Status	Unresolved

Description

The contract creates a leaf by using the `msg.sender` and the `allowedMints` in order to use for verification in `validateMerkleProof`. However, users are able to use the same parameters to recreate the same leaf and as a result it can be used in different phases. Additionally, the leaf can be reused in different chains if the same code is deployed in multiple networks.

```
function validateMerkleProof(
    bytes32[] calldata merkleProof,
    bytes32 merkleRoot,
    uint256 allowedMints
) internal view {
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender,
allowedMints));
    bool valid = MerkleProof.verify(merkleProof, merkleRoot, leaf);
    if (!valid) {
        revert InvalidProof();
    }
}
```

Recommendation

The team should restructure the validation of the proof. This can be achieved by implementing state variables to ensure that users can only `mint` once per leaf. Additionally, the leaf could be constructed by using extra parameters such as the chain id and index of the phase.

ST - Stops Transactions

Criticality	Minor / Informative
Location	audit.sol#L160,285
Status	Unresolved

Description

Transfers of the tokens are permanently disabled. This results in users not being able to trade their tokens, effectively mimicking the behavior of a soulbound token. By disabling transfers, the tokens become non-transferable and permanently linked to the holder, except in the case of burning them.

```
function _update(address to, uint256 tokenId, address auth) internal
override(ERC721, ERC721Pausable, ERC721Enumerable) returns (address) {
    if (_ownerOf(tokenId) != address(0) && to != address(0)) {
        revert TransferNotAllowed();
    }
    return super._update(to, tokenId, auth);
}
```

```
function transferFrom(address from, address to, uint256 tokenId) public
virtual override(ERC721, IERC721) {
    revert TransferNotAllowed();
}
```

Recommendation

The team should consider that unique assets may hold significant value, which could be inaccessible in the current contract state due to the limitations on transfers.

TUUC - Token URI Update Concern

Criticality	Minor / Informative
Location	audit.sol#L144
Status	Unresolved

Description

The owner is able to change the token URI. This will allow them to change the metadata of the token like the image or other critical configurations outside of the smart contract.

```
function setBaseURI(string memory _baseUri) external onlyOwner(){
    baseUri=_baseUri;
}
```

Recommendation

The team should consider that the authority to change the token URI may result in mistrust of users towards the protocol. The team may consider implementing a more decentralized approach of altering the metadata of the token.

UEV - Unrefunded Excess Value

Criticality	Minor / Informative
Location	audit.sol#L174,199
Status	Unresolved

Description

The contract does not refund users for the excess amount of the native currency they pay. This may result in users paying more for the amount of tokens they are purchasing than they should.

```
function mint(uint256 phaseIndex, bytes32[] calldata merkleProof,
uint256 allowedMints, uint256 quantity) external payable
isValidPhase(phaseIndex) isNotMaxQuantity(quantity)
hasSupply(phaseIndex, quantity)  whenNotPaused nonReentrant {
    //...
    if(msg.value < phase.mintPrice * quantity){
        revert InsufficientPayment();
    }
    //...
}
```

Recommendation

The team is advised to implement a refund mechanism to ensure that users only pay the amount necessary for purchasing the tokens.

URSV - Unused Royalty State Variables

Criticality	Minor / Informative
Location	audit.sol#L58,59
Status	Unresolved

Description

The contract defines variables intended to support a royalty mechanism, specifically `royaltyReceiver` and `royaltyPercentage`, which are set in the constructor and via the `setDefaultRoyalty` method. However, these variables are not utilized within the contract. As a result, the royalty mechanism is not implemented, and the associated declarations are redundant.

```
address public royaltyReceiver;
uint256 public royaltyPercentage;
```

Recommendation

The team should revise the royalty mechanism to ensure it aligns with the specifications. Specifically, the team is advised to revisit the ERC-2981 standard . Additionally, the royalty percentage should be validated to ensure it does not exceed the appropriate limits.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	audit.sol#L42
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.

```
bool public reallocated
```

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	audit.sol#L144
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.

```
string memory _baseUri
```

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/stable/style-guide.html#naming-conventions.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	audit.sol#L148,291
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 _baseURI() internal view override returns (string memory) {
        return baseUri;
    }

function _increaseBalance(address account, uint128 value)
        internal
        override(ERC721, ERC721Enumerable)
    {
        super._increaseBalance(account, value);
    }
```

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.

L19 - Stable Compiler Version

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

Description

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

```
pragma solidity ^0.8.22;
```

Recommendation

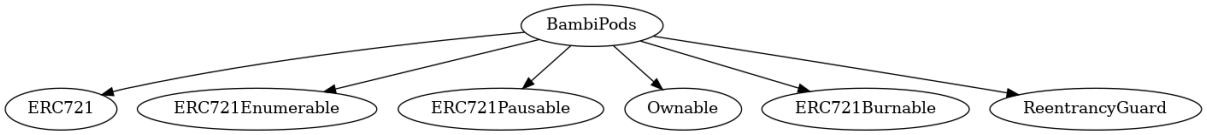
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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**
BambiPods	Implementation	ERC721, ERC721Enumerable, ERC721Pausable, Ownable, ERC721Burnable, ReentrancyGuard		
		Public	✓	ERC721 Ownable isValidRoyalty
	setDefaultRoyalty	External	✓	onlyOwner isValidRoyalty
	setBaseURI	External	✓	onlyOwner
	_baseURI	Internal		
	pause	Public	✓	onlyOwner
	unpause	Public	✓	onlyOwner
	transferFrom	Public	✓	-
	setMintPhases	External	✓	onlyOwner
	mint	External	Payable	isValidPhase isNotMaxQuantity hasSupply whenNotPaused nonReentrant
	validateMerkleProof	Internal		
	reallocatePendingSupplyToLastPhase	External	✓	onlyOwner
	withdraw	Public	✓	onlyOwner nonReentrant
	normalWithdraw	Public	✓	onlyOwner nonReentrant

	_update	Internal	✓	
	_increaseBalance	Internal	✓	
	tokenURI	Public		isValidTokenId
	supportsInterface	Public		-
		External	Payable	-
		External	Payable	-

Inheritance Graph



Summary

Loomia contract implements an nft mechanism. This audit investigates security issues, business logic concerns and potential improvements.

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 TAC 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.

A **TAC Security** Company

The Cyberscope team

cyberscope.io