Cyberscope

Audit Report
Blank

Aug 2023

Repository	https://github.com/blank-development/mintdash-contracts/tree/master
Commit	01f3042bf60f08185f3865cc78cda7601d07877a
Audited by	© cyberscope

Table of Contents

Review

Repository	https://github.com/blank-development/mintdash-contracts/tree/master
Commit	01f3042bf60f08185f3865cc78cda7601d07877a

Audit Updates

Initial Audit	28 Aug 2023

Source Files

Filename	SHA256
ERC721StakingProxy.sol	771232af78f5ef0326b785635d961e8920610f1c2a56dd4730ee31b46cb5e04a
ERC721StakingImplementation.sol	4818c5b5270c3f9af44b721d184c8a1931639db9dfae9f67c7706a2db1f6063e
ERC721Proxy.sol	67b3a477751dfc2c952b274241d67e43e7502361b007a1cad65c7b9ccc563c18
ERC721DropSignatureMintImplementation.sol	07301f1d6e3283ee230909de1366886acd32108d79ca2f4f3fe1fc7e4e11645b
ERC721DropImplementation.sol	7bff1323d1c976d3625e9b6dda74345d2a9bdded360316a6150367e95e995daf
ERC721CollectionImplementation.sol	8416b798ce137359693064230b5eb0304ae0d407744af029218cd8e20d10f789
ERC20Proxy.sol	718982e41926584bf9bbb077b5d552748f0e9a7b7b40eb578a0a62fb026af875

ERC20Implementation.sol	b6ead76fceb808060ba3c225e23d589feceac39a8986cdcaed53485f0ef2f957
ERC1155Proxy.sol	e099d6655e7ecc21134591550293d966b140349382f93b191fa9f91752b9913a
ERC1155EditionsImplementation.sol	c41521e316ba782a8103233312ba6f9f371dbc67c870071625d7dbf5bf8bf9db
lib/DropStructs.sol	b4540a066192516fc40c2e569574410876f9290d7a3d9a330d54915d156d05ec
interface/IERC721StakingImplementation.sol	4de754d6d99d5d7efe766024e71e9f61237f21d4eb78c600052a309c2a276676
interface/IERC721DropSignatureMintImplementation.sol	273e1f53249638ce05ab1f8a35b1e0937234dd2382aab0dfd08e1c944e7d90a9
interface/IERC721DropImplementation.sol	226bea05187a8d5e2cc2dfc5db1ba55b4bb7ef5573cc5940e8f116d25234277c
interface/IERC721CollectionImplementation.sol	f934b5014a68594814577cc4527b07f10bfbea99f33ce3ac9b7f160665744689
interface/IERC1155EditionsImplementation.sol	548483b5a01f68a714aed82b9da43a56d77f23b273fac5fc83922b35a61bdc94
core/Payout.sol	5fab04ca1ceac66fbe4f593fb4137f09a0d4181d1436ec3399317df1d058ec04
core/ERC721DropMetadata.sol	b8b1bb52469f8d14de8478a31faaae1187a9758f7bcadbea947482caff907aa4
core/ERC721CollectionMetadata.sol	ae6470f8cc4821d26c4285dfb95503970cfb419d49a7fc6752ac439bdd077d43
core/ERC1155Metadata.sol	f1559ca1b1d8fef4482c49a15d335c8743ee629ba0cd7bef4a5609594d90d5ae
core/AdministratedUpgradeable.sol	a5ed7ce48d4b59ae73a289e4407c78ec4adf0641a76dab8df591b5b80d7e91e3

core/interface/IPayout.sol	34b1a18f567f107722cb3cdb71902af68cd16d56ffd652b081f2858dc9dc216b
core/interface/IERC721DropMetadata.sol	d5ddebfca0d1005bd25c8892df34d9a8e37dffb78c491678af6ecb5d17139f71
core/interface/IERC721CollectionMetadata.sol	6b3c4dd0b0e3fd8b4b0a0a3ee264615373c1071a23b0ebf3f57c9d488179bce5
core/interface/IERC1155Metadata.sol	697bbe9bc673a05015d4917125bdbc2ddcc50bac88445576e902182c03fdd123

Overview

Token Functionality

ERC20Implementation

The contract implements a token mechanism which is initialized with specific attributes like name, symbol, and initial supply. It incorporates features for token burning and administrative control, while also being upgradeable for future enhancements. The contract serves as the foundational structure for a customizable and manageable digital asset.

NFT 721 and ERC 1155 Implementation

ERC721DropImplementation & ERC1155EditionsImplementation

The contract serves as an NFT implementation for ERC-721 and ERC-1155 drops of NFTs. Both contracts implement the same functionalities, with the only difference being the token base.

Minting Process

The contracts offer three distinct minting processes: `mintPublic` , `mintAllowlist` , and `mintTokenGated` . Each of these functions includes similar checks and calculations, ensuring that the minter is authorized, the corresponding mint stage is active, the mint quantity is within limits, and sufficient ETH is provided. The `mintPublic` function is open to everyone, while `mintAllowlist` is restricted to a predefined list of addresses set by the `onlyOwnerOrAdministrator` . The `mintTokenGated` function allows specific NFT holders to mint tokens. These eligible NFT holders are also predefined by the contract owner.

The contract also includes functionalities like `updatePublicMintStage` , `updateAllowlistMintStage` , `updateTokenGatedMintStage` , and `updateConfiguration` . These functions grant the `onlyOwnerOrAdministrator` the authority to modify essential parameters of the contract.

NFT 721 Signature Minting

ERC721DropSignatureMintImplementation

The contract serves as an NFT minting platform with a focus on secure and verified minting through digital signatures. It allows for a specialized minting process, `mintSigned` , which requires a valid EIP-712 signature to mint tokens. This signature ensures that only authorized minters can create new tokens under specific conditions. The contract also provides the administrator with the flexibility to update various configurations, including allowed signers for minting, maximum supply, base URI, and royalties.

Minting Process

The `mintSigned` function provides a secure way to mint new tokens. It performs several checks to validate the minting request, such as ensuring the minter is authorized, checking if the minting stage is active, and verifying the provided funds. Additionally, it uses a unique digest to prevent the same signature from being used more than once, enhancing the security of the minting process. The function allows for a high degree of customization by accepting various parameters that can be set by the contract administrator.

Staking Functionality

ERC721StakingImplementation

The contract provides a robust staking mechanism for NFTs, allowing users to stake their NFT tokens and earn rewards over time. The contract is designed to be highly secure and efficient, offering a range of features and administrative controls.

Staking Process

Users can stake their NFTs by calling the stakeNft function and providing the token ID they own. Once the NFT is staked, reward tokens begin to accumulate based on the elapsed time from when the NFT was staked until it is unstaked. The unstakeNft function allows users to unstake their NFTs, updating their unclaimed rewards and timestamps in the process. Users can claim their rewards by invoking the claimRewards function.

Administrative Controls

The contract grants the owner substantial authority over its configurations. The owner has the authority to deposit reward tokens into the contract, adjust the rate at which rewards are distributed using the `updateRewardRate` function, enable or disable the staking functionality, and deposit or withdraw ERC20 tokens. Importantly, the owner bears the responsibility of enabling rewards by transitioning to a new reward phase.

Administrative Controls and Functionality

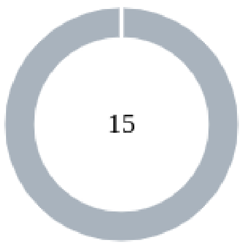
ERC721CollectionImplementation

The contract is designed to manage a collection of ERC721 tokens with additional administrative features. The contract is initialized with a name, symbol, and an administrator address. It extends functionalities from OpenZeppelin's upgradeable ERC721 and ERC2981 contracts, providing standard NFT features along with royalty support.

The administrative controls are centralized, vested primarily in the owner or an administrator. They have the authority to mint new tokens through the `mint` and `batchMint` functions, which allow for the creation of single or multiple tokens, respectively. Additionally, the owner or administrator can update the royalties associated with the NFTs using the `updateRoyalties` function, which sets a default royalty receiver and the fee's numerator. They can also change the base URI of the NFT collection through the `updateBaseURI` function, which is essential for updating the metadata associated with the tokens.

The contract also supports multiple interfaces, ensuring compatibility with ERC2981 for royalties and the standard ERC721 for NFT functionalities. Overall, the contract combines NFT capabilities with a layer of administrative control, allowing for flexible management of the NFT collection.

Findings Breakdown



🔴	Critical	0
🟠	Medium	0
●	Minor / Informative	15

Severity		Unresolved	Acknowledged	Resolved	Other
🔴	Critical	0	0	0	0
🟠	Medium	0	0	0	0
●	Minor / Informative	15	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CR	Centralization Risk	Unresolved
●	LMF	Limited Minting Flexibility	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	CI	Code Inconsistency	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	ILM	Inefficient Looping Mechanism	Unresolved
●	MU	Modifiers Usage	Unresolved
●	MPC	Merkle Proof Centralization	Unresolved
●	MSV	Missing Supply Validation	Unresolved
●	DTO	Data Type Optimization	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L16	Validate Variable Setters	Unresolved

●	L20	Succeeded Transfer Check	Unresolved

CR - Centralization Risk

Criticality	Minor / Informative
Location	ERC721StakingImplementation.sol#L29,81
Status	Unresolved

Description

The contract poses a centralization risk. Specifically, the `onlyOwnerOrAdministrator` modifier grants the owner and designated administrators the authority to supply the initial reward tokens, add or remove reward tokens, and enable rewards by updating the current phase. This concentration of power in a limited number of entities undermines the decentralized nature of the contract and poses a risk in terms of censorship, manipulation, and single points of failure.

```
function initialize(address _nftContract, address _rewardToken,
uint256 rewardsPerSecond)
        external
        initializer
    {
        __Ownable_init();

        nftToken = _nftContract;
        rewardToken = _rewardToken;

        _createNewRewardPhase(rewardsPerSecond);
    }

    function updateRewardRate(uint256 rewardsPerSecond)
external onlyOwnerOrAdministrator {

        RewardPhase storage currentRewardPhase =
rewardPhases[nextRewardPhaseId - 1];

        if (currentRewardPhase.rewardsPerSecond ==
rewardsPerSecond) {
            revert SameRewardRate();
        }

        currentRewardPhase.endTimestamp =
uint128(block.timestamp);

        _createNewRewardPhase(rewardsPerSecond);

        emit RewardRateUpdated(rewardsPerSecond);
    }
```

Recommendation

The team should carefully manage the private keys of the owner’s account or the administrator's addresses. We strongly recommend implementing a powerful security mechanism that will prevent a single user from accessing the contract's `onlyOwnerOrAdministrator` functions. This could include hardware wallets and multi-factor authentication. It is recommended to implement a more decentralized governance mechanism to distribute the authority over critical contract functions. This could involve creating a multi-signature wallet or using a timelock for critical functions.

LMF - Limited Minting Flexibility

Criticality	Minor / Informative
Location	ERC721DropSignatureMintImplementation.sol#L88
Status	Unresolved

Description

The contract is using the `mintSigned` function to allow specific addresses to mint tokens. However, the function restricts the same user from minting more than once with the same parameters. This limitation arises from the `_getDigest` function, which calculates a digest based on the `minter` address, `mintParams`, and `salt`. If none of these three parameters change, the calculated signature will remain the same, preventing additional mints for the same user with the same parameters.

```
function mintSigned(
        address recipient,
        uint256 quantity,
        SignedMintParams calldata mintParams,
        uint256 salt,
        bytes calldata signature
    ) external payable {
        ...
        // Get the digest to verify the EIP-712 signature.
        bytes32 digest = _getDigest(
            minter,
            mintParams,
            salt
        );

        // Ensure the digest has not already been used.
        if (_usedDigests[digest]) {
            revert SignatureAlreadyUsed();
        }

        // Mark the digest as used.
        _usedDigests[digest] = true;

        // Ensure correct signer signed this message.
        _checkSigner(digest, signature);
        ...
    }
```

Recommendation

It is recommended to include a `nonce` in the parameters of the `_getDigest` function. By doing so, the signature verification can occur more than once for the same user, even with the same `mintParams`. This will allow for more flexible minting capabilities while maintaining security.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	core/Payout.sol#L25,32
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 updates the state of PayoutAddress addresses or the value of Royalties even if their current state is the same as the the one passed as an argument. As a result, the contract performs redundant storage writes.

```
   function updatePayoutAddress(
        address newPayoutAddress
    ) external onlyOwnerOrAdministrator {
        _updatePayoutAddress(newPayoutAddress);
    }

    function updateRoyalties(
        address receiver,
        uint96 feeNumerator
    ) external onlyOwnerOrAdministrator {
        _updateRoyalties(receiver, feeNumerator);
    }
```

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.

CI - Code Inconsistency

Criticality	Minor / Informative
Location	core/ERC1155Metadata.sol#L105,114core/ERC721DropMetadata.sol#L91,103
Status	Unresolved

Description

The contract is using the `_updateTokenURI` function to update the URI of a specific tokenId. However, the function lacks a check for whether `totalSupply` is zero, a check that is used inside the `_updateBaseURI` function. This produces code inconsistency, as the two functions that are conceptually similar in purpose—updating URIs, have different validation criteria.

Additionally, in the ERC1155Metadata file, the `_updateProvenanceHash` function does not ensure that the minting has started, unlike its counterpart in ERC721DropMetadata. This results in another inconsistency, as it does not check if the total supply is greater than zero before updating the `provenanceHash` variable.

These inconsistencies could lead to unexpected behavior and make the code harder to maintain and understand.

```
    function _updateTokenURI(
        uint256 tokenId,
        string calldata newUri
    ) internal {
        tokenURIs[tokenId] = newUri;

        emit TokenURIUpdated(tokenId, newUri);
    }
```

```
function _updateBaseURI(
        string calldata newUri
    ) internal {
        baseURI = newUri;

        if (totalSupply() != 0) {
            emit BatchMetadataUpdate(1, _nextTokenId() - 1);
        }

        emit BaseURIUpdated(newUri);
    }
```

```
   function _updateProvenanceHash(
        bytes32 newProvenanceHash
    ) internal {
        provenanceHash = newProvenanceHash;

        emit ProvenanceHashUpdated(newProvenanceHash);
    }

    function _updateProvenanceHash(
        bytes32 newProvenanceHash
    ) internal {
        // Ensure mint did not start
        if (_totalMinted() > 0) {
            revert
ProvenanceHashCannotBeUpdatedAfterMintStarted();
        }

        provenanceHash = newProvenanceHash;

        emit ProvenanceHashUpdated(newProvenanceHash);
    }
```

Recommendation

It is recommended to add proper checks inside the `_updateTokenURI` function to align it with the `_updateBaseURI` function. Specifically, since `_updateBaseURI` includes a check that requires the total supply to be non-zero, a similar check should be implemented inside `_updateTokenURI`. Similarly, the `_updateProvenanceHash` in ERC1155Metadata should include a check to ensure that minting has started, similar with the ERC721DropMetadata.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	core/ERC1155Metadata.sol#L83ERC721StakingImplementation.sol#L95
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 updatePayer(
        address payer,
        bool isAllowed
    ) external onlyOwnerOrAdministrator {
        allowedPayers[payer] = isAllowed;
    }
```

```
    function toggleStaking() external onlyOwnerOrAdministrator {
        isStakingEnabled = !isStakingEnabled;
    }
```

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.

ILM - Inefficient Looping Mechanism

Criticality	Minor / Informative
Location	core/ERC1155Metadata.sol#L38
Status	Unresolved

Description

The contract is using two separate `for` loops within the `airdrop` function to iterate through the `recipients.length` and `tokenId.length` . The first loop is responsible for minting tokens, while the second loop checks if the total supply exceeds the maximum supply for each tokenId. This approach adds unnecessary complexity to the code and is gas-inefficient, as it requires two separate iterations through the arrays.

```
function airdrop(
        address[] calldata to,
        uint256[] calldata tokenId,
        uint64[] calldata quantity
    ) external onlyOwnerOrAdministrator {
        address[] memory recipients = to;

        for (uint64 i = 0; i < recipients.length; ) {
            _mint(recipients[i], tokenId[i], quantity[i], "");

            unchecked {
                ++i;
            }
        }

        for (uint64 i = 0; i < tokenId.length; ) {
            if (totalSupply[tokenId[i]] >
maxSupply[tokenId[i]]) {
                revert MintQuantityExceedsMaxSupply();
            }

            unchecked {
                ++i;
            }
        }
    }
```

Recommendation

It is recommended to simplify the looping mechanism by implementing a single `for` loop that handles both the minting and the supply check. This would make the code easier to read and maintain, as well as more gas-efficient. Alternatively, since the `tokenId` are unique, it is recommended to loop only through the unique `tokenId` to prevent redundant iterations.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	ERC721DropImplementation.sol#L68core/Payout.sol#L45,71,89
Status	Unresolved

Description

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

```
if (tx.origin != msg.sender) {
    revert PayerNotAllowed();
}
```

```
if (address(this).balance == 0) {
    revert NothingToWithdraw();
}

if (payoutAddress == address(0)) {
    revert InvalidPayoutAddress();
}

if (platformFeesAddress == address(0)) {
    revert InvalidPlatformFeesAddress();
}

if (newPayoutAddress == address(0)) {
    revert PayoutAddressCannotBeZeroAddress();
}


if (newPlatformFeesAddress == address(0)) {
    revert PlatformFeesAddressCannotBeZeroAddress();
}
```

Recommendation

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

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	ERC721DropImplementation.sol#L81ERC1155EditionsImplementation.sol#L92
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 mintAllowlist(
        uint256 allowlistStageId,
        address recipient,
        uint256 quantity,
        bytes32[] calldata merkleProof
    ) external payable {
    ...
        if (
            !MerkleProof.verifyCalldata(
                merkleProof,
                allowlistMintStage.merkleRoot,
                keccak256(abi.encodePacked(minter))
            )
        ) {
            revert AllowlistStageInvalidProof();
        }
    ...
}
```

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. Some suggestions are:

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.
- Renouncing the ownership will eliminate the threats but it is non-reversible.

MSV - Missing Supply Validation

Criticality	Minor / Informative
Location	core/ERC721DropMetadata.sol#L77core/ERC1155Metadata.sol#L91
Status	Unresolved

Description

The contract is using the `_updateMaxSupply` function to modify the `maxSupply` associated with the ERC721 or ERC1155 token implementation. However, the function does not verify if the `newMaxSupply` being set is greater than or equal to the current total supply of tokens. This means that the `_updateMaxSupply` function could potentially set `maxSupply` to a value lower than the existing total supply for that tokenId, leading to inconsistencies and potential issues in token management.

```
    function _updateMaxSupply(
        uint256 newMaxSupply
    ) internal {
        // Ensure the max supply does not exceed the maximum
value of uint64.
        if (newMaxSupply > 2 ** 64 - 1) {
            revert CannotExceedMaxSupplyOfUint64();
        }

        maxSupply = newMaxSupply;

        emit MaxSupplyUpdated(newMaxSupply);
    }
     function _updateMaxSupply(
        uint256 tokenId,
        uint256 newMaxSupply
    ) internal {
        ...
    }
```

Recommendation

It is recommended to introduce a proper check within the `_updateMaxSupply` function to prevent `maxSupply` from being set to a value lower than the current total supply of tokens. This can be achieved by adding an if statement that does not allow the `maxSupply` to be set lower than the current total supply.

DTO - Data Type Optimization

Criticality	Minor / Informative
Location	core/ERC721DropMetadata.sol#L81core/ERC1155Metadata.sol#L96
Status	Unresolved

Description

The contract is currently using an `if` statement to check if the `newMaxSupply` exceeds `2 ** 64 - 1`. Specifically, while the functions `_updateMaxSupply` contain the an if statement to prevent `newMaxSupply` from exceeding `2 ** 64 - 1` , they introduce unnecessary computational overhead and complexity. Since the maximum allowable value for `newMaxSupply` is `2 ** 64 - 1`, using a `uint64` data type for `newMaxSupply` would inherently enforce this constraint without the need for an explicit `if` statement.

```
    function _updateMaxSupply(
        uint256 newMaxSupply
    ) internal {
        // Ensure the max supply does not exceed the maximum
value of uint64.
        if (newMaxSupply > 2 ** 64 - 1) {
            revert CannotExceedMaxSupplyOfUint64();
        }
        ...
    }
```

```
function _updateMaxSupply(
        uint256 tokenId,
        uint256 newMaxSupply
    ) internal {
        // Ensure the max supply does not exceed the maximum
value of uint64.
        if (newMaxSupply > 2 ** 64 - 1) {
            revert CannotExceedMaxSupplyOfUint64();
        }
        ..
    }
```

Recommendation

It is recommended to use `uint64` for the `newMaxSupply` variable instead of performing an explicit check with an `if` statement. By using `uint64`, the Solidity type system will automatically ensure that `newMaxSupply` cannot be set to a value greater than `2 ** 64 - 1` by eliminating the need for the `if` statement and reducing computational overhead.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	ERC721StakingImplementation.sol#L29ERC721DropSignatureMintImplementation.sol#L30,31,32,33,34,35,36,39,40,41ERC721DropImplementation.sol#L41,42,43ERC721CollectionImplementation.sol#L20ERC20Implementation.sol#L15,16ERC1155EditionsImplementation.sol#L45core/Payout.sol#L17core/AdministratedUpgradeable.sol#L18
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.

```
address _nftContract
address _rewardToken
uint256 internal _CHAIN_ID
bytes32 internal _SIGNED_MINT_TYPEHASH
bytes32 internal _MINT_PARAMS_TYPEHASH
bytes32 internal _EIP_712_DOMAIN_TYPEHASH
bytes32 internal _NAME_HASH
bytes32 internal _VERSION_HASH
bytes32 internal _DOMAIN_SEPARATOR
string memory _name
string memory _symbol
address _administrator
string calldata _name
string calldata _symbol

...
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	ERC721DropImplementation.sol#L31ERC1155EditionsImplementation.sol #L36
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
mapping(address  => mapping(address  => mapping(uint256  =>
bool )))
        private _tokenHolderRedeemed

mapping(uint256  =>
        mapping(address  => mapping(address  => mapping(uint256
=> bool )))) private _tokenHolderRedeemed
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	core/AdministratedUpgradeable.sol#L18
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 __Administrated_init(address _administrator)
     internal
     onlyInitializing
    {
        administrator = _administrator;
    }
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	ERC721StakingImplementation.sol#L35,36
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.

```
nftToken = _nftContract
rewardToken = _rewardToken
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	ERC721StakingImplementation.sol#L76,100,107
Status	Unresolved

Description

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

```
IERC20(rewardToken).transfer(msg.sender, rewards)
IERC20(rewardToken).transferFrom(sender, address(this), amount)
IERC20(rewardToken).transfer(recipient, amount)
```

Recommendation

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

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ERC721Staking Implementation	Implementation	AdministratedUpgradeable, MulticallUpgradeable, IERC721StakingImplementation		
	initialize	External	✓	initializer
	stakeNft	External	✓	-
	unstakeNft	External	✓	-
	claimRewards	External	✓	-
	updateRewardRate	External	✓	onlyOwnerOrAdministrator
	toggleStaking	External	✓	onlyOwnerOrAdministrator
	depositERC20Tokens	External	✓	onlyOwnerOrAdministrator
	withdrawERC20Tokens	External	✓	onlyOwnerOrAdministrator
	getStakerUnclaimedRewards	External		-
	_stakeNft	Internal	✓	
	_unstakeNft	Internal	✓	
	_updateStakerUnclaimedRewards	Internal	✓	
	_updateStakerTimestamps	Internal	✓	
	_createNewRewardPhase	Internal	✓	

	_calculateRewardsSinceLastUpdate	Internal		
	_getElapsedPhaseTime	Internal		
	_calculatePhaseRewards	Internal		
ERC721DropSignatureMintImplementation	Implementation	AdministratedUpgradeable, ERC721DropMetadata, Payout, IERC721DropSignatureMintImplementation		
	initialize	External	✓	initializerERC721A initializer
	mintSigned	External	Payable	-
	updateAllowedSigner	External	✓	onlyOwnerOrAdministrator
	updateConfiguration	External	✓	onlyOwnerOrAdministrator
	supportsInterface	Public		-
	_checkSigner	Internal		
	_getDigest	Internal		
	_domainSeparator	Internal		
	_deriveDomainSeparator	Internal		
ERC721DropImplementation	Implementation	AdministratedUpgradeable, ERC721DropMetadata, Payout, IERC721DropImplementation		
	initialize	External	✓	initializerERC721A initializer

	mintPublic	External	Payable	-
	mintAllowlist	External	Payable	-
	mintTokenGated	External	Payable	-
	getTokenGatedIsRedeemed	External		-
	updatePublicMintStage	External	✓	onlyOwnerOrAdministrator
	updateAllowlistMintStage	External	✓	onlyOwnerOrAdministrator
	updateTokenGatedMintStage	External	✓	onlyOwnerOrAdministrator
	updateConfiguration	External	✓	onlyOwnerOrAdministrator
	supportsInterface	Public		-
	_updatePublicMintStage	Internal	✓	
	_updateAllowlistMintStage	Internal	✓	
	_updateTokenGatedMintStage	Internal	✓	
ERC721CollectionImplementation	Implementation	AdministratedUpgradeable, ERC721CollectionMetadata, ERC2981Upgradeable, IERC721CollectionImplementation		
	initialize	External	✓	initializer
	mint	External	✓	onlyOwnerOrAdministrator
	batchMint	External	✓	onlyOwnerOrAdministrator
	updateRoyalties	External	✓	onlyOwnerOrAdministrator

	updateBaseURI	External	✓	onlyOwnerOrAdministrator
	supportsInterface	Public		-
ERC20Implementation	Implementation	ERC20BurnableUpgradeable, AdministratedUpgradeable, MulticallUpgradeable		
	initialize	External	✓	initializer
ERC1155EditionsImplementation	Implementation	AdministratedUpgradeable, ERC1155Metadata, Payout, IERC1155EditionsImplementation		
	initialize	External	✓	initializer
	mintPublic	External	Payable	-
	mintAllowlist	External	Payable	-
	mintTokenGated	External	Payable	-
	getPublicMintStage	External		-
	getAllowlistMintStage	External		-
	getTokenGatedMintStage	External		-
	getTokenGatedIsRedeemed	External		-
	updateConfiguration	External	✓	onlyOwnerOrAdministrator
	updatePublicMintStage	External	✓	onlyOwnerOrAdministrator

	updateAllowlistMintStage	External	✓	onlyOwnerOrAdministrator
	updateTokenGatedMintStage	External	✓	onlyOwnerOrAdministrator
	supportsInterface	Public		-
	_updatePublicMintStage	Internal	✓	
	_updateAllowlistMintStage	Internal	✓	
	_updateTokenGatedMintStage	Internal	✓	
IERC721StakingImplementation	Interface			
	stakeNft	External	✓	-
	unstakeNft	External	✓	-
	claimRewards	External	✓	-
	updateRewardRate	External	✓	-
	toggleStaking	External	✓	-
	depositERC20Tokens	External	✓	-
	withdrawERC20Tokens	External	✓	-
	getStakerUnclaimedRewards	External		-
IERC721DropSignatureMintImplementation	Interface			
	mintSigned	External	Payable	-
	updateConfiguration	External	✓	-

IERC721DropImplementation	Interface			
	mintPublic	External	Payable	-
	mintAllowlist	External	Payable	-
	mintTokenGated	External	Payable	-
	getTokenGatedIsRedeemed	External		-
	updateConfiguration	External	✓	-
	updatePublicMintStage	External	✓	-
	updateAllowlistMintStage	External	✓	-
	updateTokenGatedMintStage	External	✓	-
IERC721CollectionImplementation	Interface			
IERC1155EditionsImplementation	Interface			
	mintPublic	External	Payable	-
	mintAllowlist	External	Payable	-
	mintTokenGated	External	Payable	-
	getAllowlistMintStage	External	✓	-
	getTokenGatedMintStage	External	✓	-
	getTokenGatedIsRedeemed	External		-
	updateConfiguration	External	✓	-
	updatePublicMintStage	External	✓	-
	updateAllowlistMintStage	External	✓	-

	updateTokenGatedMintStage	External	✓	-
Payout	Implementation	AdministratedUpgradeable, ERC2981Upgradeable, IPayout		
	__Payout_init	Internal	✓	onlyInitializing
	updatePlatformFees	External	✓	onlyAdministrator
	updatePayoutAddress	External	✓	onlyOwnerOrAdministrator
	updateRoyalties	External	✓	onlyOwnerOrAdministrator
	withdrawAllFunds	External	✓	onlyOwnerOrAdministrator
	_updatePayoutAddress	Internal	✓	
	_updateRoyalties	Internal	✓	
	_updatePlatformFees	Internal	✓	
ERC721DropMetadata	Implementation	AdministratedUpgradeable, ERC721AUpgradeable, MulticallUpgradeable, IERC721DropMetadata		
	getAmountMinted	External		-
	burn	External	✓	-
	airdrop	External	✓	onlyOwnerOrAdministrator
	updateMaxSupply	External	✓	onlyOwnerOrAdministrator

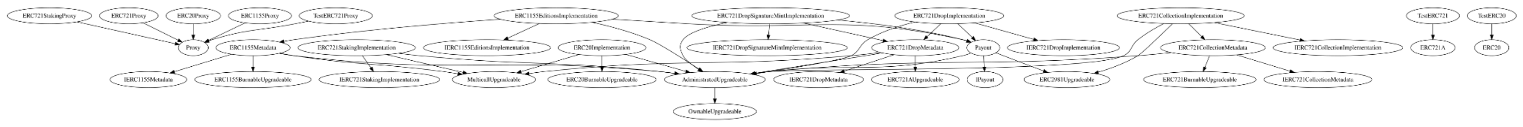
	updateBaseURI	External	✓	onlyOwnerOrAdministrator
	updateProvenanceHash	External	✓	onlyOwnerOrAdministrator
	updatePayer	External	✓	onlyOwnerOrAdministrator
	_updateMaxSupply	Internal	✓	
	_updateBaseURI	Internal	✓	
	_updateProvenanceHash	Internal	✓	
	_checkPayer	Internal		
	_checkFunds	Internal		
	_checkMintQuantity	Internal		
	_checkStageActive	Internal		
	_mintBase	Internal	✓	
	_baseURI	Internal		
	_startTokenId	Internal		
ERC721CollectionMetadata	Implementation	AdministratedUpgradeable, ERC721BurnableUpgradeable, IERC721CollectionMetadata		
	tokenURI	Public		-
	totalSupply	External		-
	_mintBase	Internal	✓	
	_batchMintBase	Internal	✓	
	_burn	Internal	✓	

	_baseURI	Internal		
ERC1155Metadata	Implementation	AdministratedUpgradeable, ERC1155BurnableUpgradeable, MulticallUpgradeable, IERC1155Metadata		
	uri	Public		-
	getAmountMinted	External		-
	airdrop	External	✓	onlyOwnerOrAdministrator
	updateMaxSupply	External	✓	onlyOwnerOrAdministrator
	updateTokenURI	External	✓	onlyOwnerOrAdministrator
	updateProvenanceHash	External	✓	onlyOwnerOrAdministrator
	updatePayer	External	✓	onlyOwnerOrAdministrator
	_updateMaxSupply	Internal	✓	
	_updateTokenURI	Internal	✓	
	_updateProvenanceHash	Internal	✓	
	_checkPayer	Internal		
	_checkFunds	Internal		
	_checkMintQuantity	Internal		
	_checkStageActive	Internal		
	_mintBase	Internal	✓	

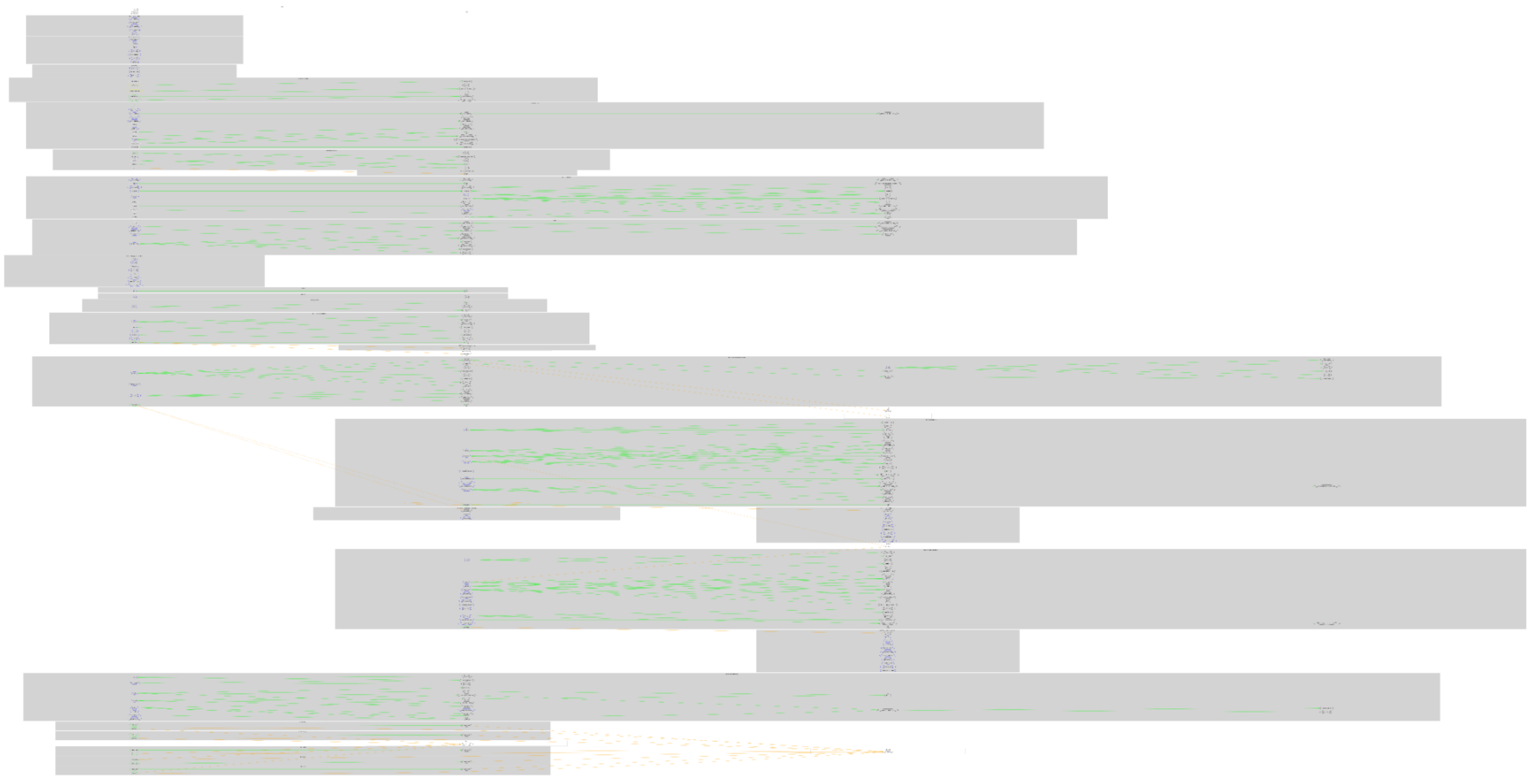
AdministratedUpgradeable	Implementation	OwnableUpgradeable		
	__Administrated_init	Internal	✓	onlyInitializing
	renounceAdministration	Public	✓	onlyAdministrator
	transferAdministration	Public	✓	onlyAdministrator
	_transferAdministration	Internal	✓	
IPayout	Interface			
	updateRoyalties	External	✓	-
	withdrawAllFunds	External	✓	-
	updatePayoutAddress	External	✓	-
IERC721DropMetadata	Interface			
	getAmountMinted	External		-
	airdrop	External	✓	-
	burn	External	✓	-
	updateMaxSupply	External	✓	-
	updateProvenanceHash	External	✓	-
	updateBaseURI	External	✓	-
	updatePayer	External	✓	-
IERC721CollectionMetadata	Interface			

IERC1155Metadata	Interface			
	getAmountMinted	External		-
	updateMaxSupply	External	✓	-
	updateProvenanceHash	External	✓	-
	updateTokenURI	External	✓	-
	updatePayer	External	✓	-

Inheritance Graph



Flow Graph



Summary

Blank smart contracts implement a comprehensive solution that implements token management, NFT creation, staking, and rewards distribution. This review aims to identify security vulnerabilities, assess the integrity of the business logic, and suggest potential enhancements.

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