Cyberscope

Audit Report

Crypto Homosapiens

May 2024

Network	Sepolia Testnet
Address	0xaf1Cdb7aD56ff12c21EEb9C1d03f4E90aa48D1C0
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://sepolia.etherscan.io/address/0x4a88f5d753096a7f2e0d3531a06db49a0f8e7384

Audit Updates

Initial Audit	18 May 2024 https://github.com/cyberscope-io/audits/blob/main/1-chs/v1/audit.pdf
Corrected Phase 2	28 May 2024

Source Files

Filename	SHA256
MyToken.sol	8e6fd9ef35effdb5f121b1f9e5df31eb3a51d4db900946353dd9042a420bf565

Overview

The CHAPC smart contract is an ERC721 token implementation designed to facilitate the minting, distribution, and management of non-fungible tokens (NFTs) within a structured presale and airdrop framework. The contract inherits from ERC721Enumerable and VRFConsumerBaseV2Plus, incorporating functionality from OpenZeppelin's ERC721 standards and Chainlink's Verifiable Random Function v2 (VRF) for randomness.

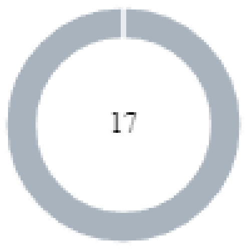
The primary functionality of the CHAPC contract is to manage the minting of NFTs across three distinct packages, each with a predefined token supply limit. The contract supports a presale phase, where only whitelisted addresses are allowed to mint tokens, verified through Merkle proof validation. Additionally, the contract includes mechanisms for airdrop winners, enabling the contract owner to mint NFTs directly to specified addresses. Furthermore, the contract owner can setAirdropWinners, and the winners will be eligible to get NFTs from packageId 0 or 1, but not 2.

The contract also integrates a token distribution mechanism, rewarding NFT transfers with tokens, once per day, allocated among the buyer, the creator, and the first owner of the token. This distribution is governed by predefined percentages and is designed to incentivize engagement and transactions within the ecosystem.

Chainlink VRF is utilized to introduce randomness in the minting process, ensuring fair and unpredictable token assignments. This is achieved through a request and fulfillment system, where random seeds are generated and used to shuffle token IDs, enhancing the randomness and security of the minting process.

Additionally, the contract includes functionality for purchasing "bones", another type of token. It provides administrative functions for the contract owner, such as pausing the contract, adjusting minting limits, and withdrawing collected Ether.

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	17

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

Diagnostics

🔴 Critical 🟠 Medium ● Minor / Informative

Severity	Code	Description	Status
●	CCR	Contract Centralization Risk	Unresolved
●	DMPA	Distribution Mechanism Potential Abuse	Unresolved
●	ICV	Inadequate Contract Verification	Unresolved
●	MPC	Merkle Proof Centralization	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	SMF	Stop Minting Functionality	Unresolved
●	UT	Unnecessary Typecasting	Unresolved
●	UOM	Unrestricted Owner Minting	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L08	Tautology or Contradiction	Unresolved

●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	MyToken.sol#L272,333,380,464,506,524,528,534,539,545,563...
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 setBoneWinners(address[] memory _winners) external
onlyOwner {
    for (uint256 i = 0; i < _winners.length; i++) {
        boneWinners[_winners[i]] = true;
    }
}

function setPhase(Phase _phase) external onlyOwner {
    require(_phase == Phase.OG || _phase == Phase.Whitelist,
"Invalid phase");
    currentPhase = _phase;
}
```

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.

DMPA - Distribution Mechanism Potential Abuse

Criticality	Minor / Informative
Location	MyToken.sol#L446
Status	Unresolved

Description

The `tokenDistribution` function is designed to distribute tokens to buyers whenever an NFT is transferred. This function is called within the overridden transferFrom function, which means that every transfer triggers the token distribution process. However, this mechanism can be exploited by malicious actors who repeatedly transfer the same NFT back and forth between addresses to artificially increase their token rewards. The lack of safeguards against such behavior allows users to game the system, potentially leading to an unfair distribution of tokens and depletion of the token pool.

```
function transferFrom(
    address from,
    address to,
    uint256 id
) public virtual override(ERC721, IERC721) {
    super.transferFrom(from, to, id);
    if (isTokenDistributing) {
        tokenDistribution(from, to, id);
    }
}
```

Recommendation

To prevent abuse of the token distribution mechanism, it is recommended to implement safeguards that limit the frequency of eligible transfers for token distribution. These measures will mitigate the risk of abuse, ensuring a fairer and more controlled distribution of tokens.

ICV - Inadequate Contract Verification

Criticality	Minor / Informative
Location	MyToken.sol#L179,246,311
Status	Unresolved

Description

The contract utilizes the `isContract` function to prevent contracts from participating in the `presaleMint` and `airdropMint` functions. While the `isContract` function can accurately determine if an address is a contract, it cannot reliably verify that an address is not a contract. Specifically, `isContract` will return false for certain types of addresses, including externally-owned accounts (EOAs), contracts in construction, addresses where contracts will be created, and addresses where contracts previously existed but have been destroyed. As a result, relying solely on `isContract` for security may create a false sense of protection, as malicious actors could exploit these scenarios to bypass the contract verification.

```
require(!isContract(msg.sender), "Contracts are not allowed to
mint");

function isContract(address account) internal view returns
(bool) {
    uint256 size;
    assembly {
        size := extcodesize(account)
    }
    return size > 0;
}
```

Recommendation

It is recommended to implement additional security measures to prevent contracts from participating in the `presaleMint` and `airdropMint` functions. While `isContract` can be a useful tool, it should not be the sole method of verification. By incorporating additional measures, the contract will be better protected against potential exploits and ensure that only legitimate users can participate in the minting process.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	MyToken.sol#L157
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 presaleMint(Cart[] memory cart, bytes32[] calldata
_merkleProof) external payable nonReentrant {
    require(!paused, "the contract is paused");

    uint256 perAddressLimit = boneWinners[msg.sender] ? 5 :
nftPerAddressLimit;
    uint256 maxLimitPS = boneWinners[msg.sender] ? 5 :
maxLimitPerSession;
    bytes32 leaf = keccak256(abi.encode(msg.sender));

    if(presaleMintOnlyOgOrWhitelisted) {
        if (currentPhase == Phase.OG) {
        require(
            MerkleProof.verify(_merkleProof, ogMerkleRoot,
leaf),
            "Not in OG list"
            );
        } else if (currentPhase == Phase.Whitelist) {
            require(
                MerkleProof.verify(_merkleProof,
whitelistMerkleRoot, leaf),
                "Not whitelisted"
            );
        }
        ...

}
```

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.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	MyToken.sol#L582
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(os)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	MyToken.sol#L506,524,528,534,539,545,563,568,574
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 setWhitelistMerkleRoot(bytes32 _newMerkleRoot)
external onlyOwner {
    whitelistMerkleRoot = _newMerkleRoot;
}

function setOgMerkleRoot(bytes32 _newMerkleRoot) external
onlyOwner {
    ogMerkleRoot = _newMerkleRoot;
}
```

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.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	MyToken.sol#L506,524,528,534,539,545,563,568,574
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
function pause(bool _state) external onlyOwner {
    paused = _state;
}

function setWhitelistMerkleRoot(bytes32 _newMerkleRoot)
external onlyOwner {
    whitelistMerkleRoot = _newMerkleRoot;
}

function setOgMerkleRoot(bytes32 _newMerkleRoot) external
onlyOwner {
    ogMerkleRoot = _newMerkleRoot;
}

...
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

SMF - Stop Minting Functionality

Criticality	Minor / Informative
Location	MyToken.sol#L157,244,272,506
Status	Unresolved

Description

The contract includes a function `pause` that allows the contract owner to pause and unpause the minting process at their discretion. While the ability to pause and unpause the contract can be useful for handling emergencies or updates, it also introduces a significant centralization risk. The contract owner has control over the minting process, which means they can halt or resume minting at any time without any checks or balances. This level of control can undermine the trust of the participants, as they must rely on the contract owner's discretion for the availability of minting.

```
function presaleMint(Cart[] memory cart, bytes32[] calldata
_merkleProof) external payable nonReentrant {
        require(!paused, "the contract is paused");
        ...
}

function airdropMint() external nonReentrant {
        require(!paused, "the contract is paused");
}

function ownerMint(Cart[] memory cart) external onlyOwner
nonReentrant {
        require(!paused, "the contract is paused");
}


function pause(bool _state) external onlyOwner {
paused = _state;
}
```

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.

UT - Unnecessary Typecasting

Criticality	Minor / Informative
Location	MyToken.sol#L464
Status	Unresolved

Description

The `revealBatchOfPackages` function contains unnecessary typecasting of the `batchSize` . The `batchSize` constant is initially declared as a `uint256` and then typecast to `uint8` before being added to the `uint16` array `lastRevealedTokenOfPackages` . This typecasting is redundant. The current implementation complicates the code without providing any additional safety or functionality. Furthermore, the unnecessary typecasting can obscure the readability of the code.

```
function revealBatchOfPackages(uint8 packageId) external
onlyOwner {
        require(
            totalSupplyOfPackages[packageId] >=
                lastRevealedTokenOfPackages[packageId] +
batchSize,
            "Insufficient total supply for package"
        );
        require(
            randomSeeds[packageId][
                lastRevealedTokenOfPackages[packageId] /
batchSize
            ] != 0,
            "Random seed not yet generated."
        );
        require(
            lastRevealedTokenOfPackages[packageId] +
                uint8(uint256(batchSize)) <=
                tokenLimitOfPackages[packageId],
            "All supplies revealed."
        );

        uint256 starts = startIdsOfPackages[packageId] +
lastRevealedTokenOfPackages[packageId];
        uint256[batchSize] memory ranges;
        for (uint256 i = 0; i < batchSize; i++) {
            ranges[i] = starts + i;
        }

        uint256 batchNumber =
lastRevealedTokenOfPackages[packageId] / batchSize;
        shuffleArray(ranges,
randomSeeds[packageId][batchNumber]);

        for (uint256 i = 0; i < batchSize; i++) {
            revealedOrder[packageId][starts + i] = ranges[i];
        }

        lastRevealedTokenOfPackages[packageId] +=
uint8(uint256(batchSize));
    }
```

Recommendation

To simplify the code and enhance readability, it is recommended to remove the redundant typecasting from `uint256` to `uint8` for the `batchSize` constant in the `revealBatchOfPackages` function, since `lastRevealedTokenOfPackages` is declared as a `uint16` array. This adjustment will ensure the code remains clean and easy to understand while maintaining its intended operation.

UOM - Unrestricted Owner Minting

Criticality	Minor / Informative
Location	MyToken.sol#L272
Status	Unresolved

Description

The contract includes an `ownerMint` function that allows the contract owner to mint tokens without any cost. Unlike the `presaleMint` function, which requires users to pay a specific amount of native currency to mint tokens, the `ownerMint` function enables the owner to bypass this payment requirement and mint tokens freely. This introduces risks of centralization and economic exploitation. The contract owner can potentially mint a very high number of tokens, leading to centralization of token ownership and undermining the fairness and integrity of the token distribution.

```
function ownerMint(Cart[] memory cart) external onlyOwner
nonReentrant {
        require(!paused, "the contract is paused");
        uint256 totalWishQuantity;

        // Calculate total price and total quantity to be
minted
        for (uint8 i = 0; i < cart.length; i++) {
            require(cart[i].packageId >= 0 && cart[i].packageId
<= 2, "Invalid package id");
            require(cart[i].quantity > 0, "need to mint at
least 1 NFT");
            require(
                totalSupplyOfPackages[cart[i].packageId] +
cart[i].quantity <= tokenLimitOfPackages[cart[i].packageId],
                "Exceeds total supply for package"
            );

            totalWishQuantity += cart[i].quantity;
        }
        require(totalSupply() + totalWishQuantity <=
tokenLimit, "Exceeds total supply");

        // Update state variables before minting
        for (uint8 i = 0; i < cart.length; i++) {
            uint256 packageId = cart[i].packageId;
            for (uint8 j = 0; j < cart[i].quantity; j++) {
                uint256 tokenId = startIdsOfPackages[packageId]
+ totalSupplyOfPackages[packageId];
                totalSupplyOfPackages[packageId] += 1;
                firstOwners[tokenId] =
FirstOwner({ownerAddress: msg.sender, collectedToken: 0});
            }
        }

        // Mint tokens after updating state variables
        for (uint8 i = 0; i < cart.length; i++) {
            uint256 packageId = cart[i].packageId;
            for (uint8 j = 0; j < cart[i].quantity; j++) {
                uint256 tokenId = startIdsOfPackages[packageId]
+ totalSupplyOfPackages[packageId] - cart[i].quantity + j;
                _safeMint(msg.sender, tokenId);
                emit OwnerMint(tokenId);
            }
        }
    }
```

Recommendation

It is recommended to implement restrictions on the `ownerMint` function to ensure fair and decentralized token distribution. Introducing limits on the number of tokens the owner can mint, requiring the owner to pay minting fees similar to those paid by regular users, or implementing governance mechanisms for significant minting actions can help protect the integrity of the token distribution and maintain the economic fairness of the ecosystem. These measures will help ensure that the minting process remains transparent, equitable, and resistant to potential exploitation by the contract owner.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	MyToken.sol#L69,70,79,84,85,86,87
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.

```
bytes32 public airdropWinnersMerkleRoot
uint32 numWords = 1
uint256 public bonePrice = 0.01 ether
uint16 requestConfirmations = 3
uint16 public maxLimitPerSession = 2
uint16 public nftPerAddressLimit = 2
uint16 public bonePerAddressLimit = 2
```

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	MyToken.sol#L61,76,92,157,370,371,506,524,528,545,550,563,568,574
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.

```
IVRFCoordinatorV2Plus COORDINATOR
uint256 constant tokenCollectLimit = 1 days
mapping(uint256 => RequestStatus) public s_requests
bytes32[] calldata _merkleProof
uint256 _requestId
uint256[] calldata  _randomWords
bool _state
bytes32 _newMerkleRoot
string memory _newBaseURI
address[] memory _winners
uint8[] memory _packageIds
bool _newValue
Phase _phase
```

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	MyToken.sol#L101
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.

```
address[] public tokenCollectors;
```

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.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	MyToken.sol#L187,278,553
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
require(cart[i].packageId >= 0 && cart[i].packageId <= 2,
"Invalid package id")
require(_packageIds[i] >= 0 && _packageIds[i] < 2, "Invalid
package id")
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	MyToken.sol#L516,518
Status	Unresolved

Description

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

```
uint256 smallerSeed = seed / divisor
uint256 j = (smallerSeed * i) % (i + 1)
```

Recommendation

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

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	MyToken.sol#L484
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[batchSize] memory ranges
```

Recommendation

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

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	MyToken.sol#L313
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)
        }
```

Recommendation

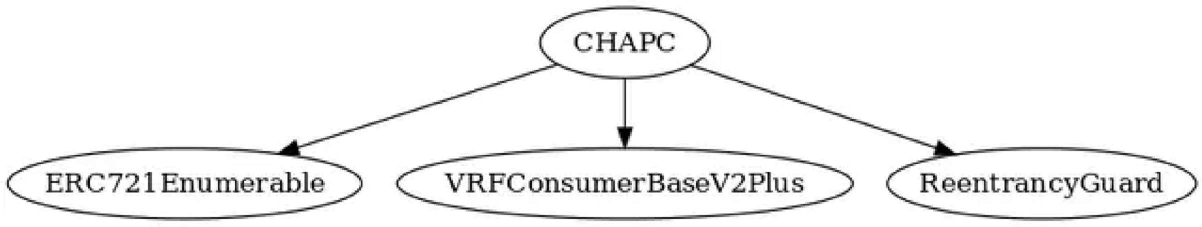
It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

Functions Analysis

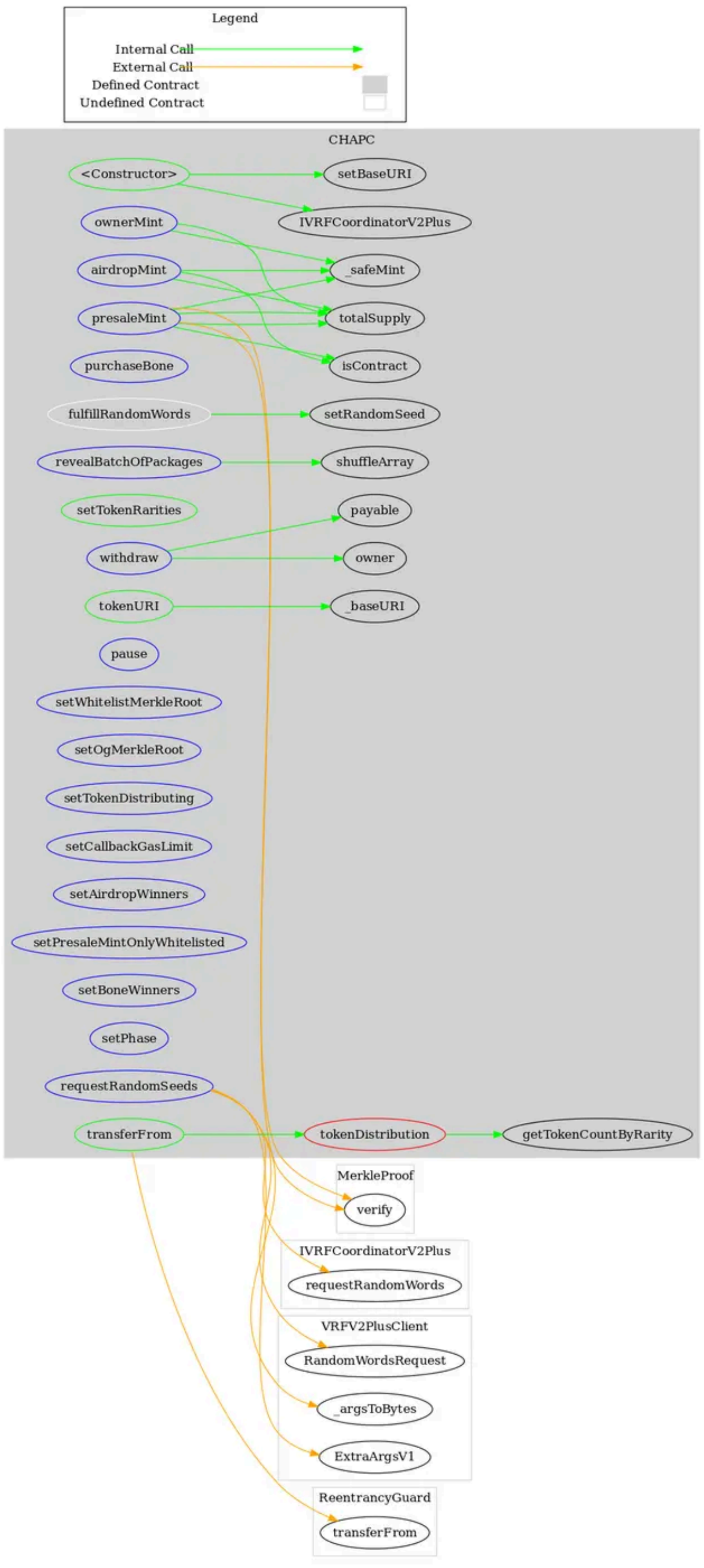
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
CHAPC	Implementation	ERC721Enumerable, VRFConsumerBaseV2Plus, ReentrancyGuard		
		Public	✓	ERC721 VRFConsumerBaseV2Plus
	presaleMint	External	Payable	nonReentrant
	airdropMint	External	✓	nonReentrant
	ownerMint	External	✓	onlyOwner nonReentrant
	isContract	Internal		
	purchaseBone	External	Payable	nonReentrant
	requestRandomSeeds	External	✓	onlyOwner
	fulfillRandomWords	Internal	✓	
	setTokenRarities	Public	✓	onlyOwner
	getTokenCountByRarity	Private		
	tokenDistribution	Private	✓	
	transferFrom	Public	✓	-
	setRandomSeed	Internal	✓	
	revealBatchOfPackages	External	✓	onlyOwner
	_baseURI	Internal		

	pause	External	✓	onlyOwner
	shuffleArray	Internal		
	setWhitelistMerkleRoot	External	✓	onlyOwner
	setOgMerkleRoot	External	✓	onlyOwner
	setTokenDistributing	External	✓	onlyOwner
	setCallbackGasLimit	External	✓	onlyOwner
	setBaseURI	Public	✓	onlyOwner
	setAirdropWinners	External	✓	onlyOwner
	setPresaleMintOnlyWhitelisted	External	✓	onlyOwner
	setBoneWinners	External	✓	onlyOwner
	setPhase	External	✓	onlyOwner
	withdraw	External	Payable	onlyOwner
	tokenURI	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
CHAPC
<Constructor>
setBaseURI
IVRFCoordinatorV2Plus
ownerMint
_safeMint
airdropMint
totalSupply
presaleMint
isContract
purchaseBone
fulfillRandomWords
setRandomSeed
revealBatchOfPackages
shuffleArray
setTokenRarities
payable
withdraw
owner
tokenURI
_baseURI
pause
setWhitelistMerkleRoot
setOgMerkleRoot
setTokenDistributing
setCallbackGasLimit
setAirdropWinners
setPresaleMintOnlyWhitelisted
setBoneWinners
setPhase
requestRandomSeeds
transferFrom
tokenDistribution
getTokenCountByRarity
MerkleProof
verify
IVRFCoordinatorV2Plus
requestRandomWords
VRFV2PlusClient
RandomWordsRequest
_argsToBytes
ExtraArgsV1
ReentrancyGuard
transferFrom

Summary

Crypto Homosapiens contract implements a NFT mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io