Cyberscope

Audit Report

Crypto Homosapiens

May 2024

Network	Sepolia Testnet
Address	0x4a88f5D753096a7f2e0D3531a06dB49A0F8E7384
Audited by	© cyberscope

Table of Contents

Review

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

Audit Updates

Initial Audit	18 May 2024

Source Files

Filename	SHA256
homosapiens.sol	c7b064c1d62e69d2325dbf1531010a9a55090cd8b0fdfc72b858a0984c7f6109
hardhat/console.sol	748b15b53c6183f4ea2ec90ff79e192ff9efde9b664a3925b44ddc5dadb7e3bd

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.

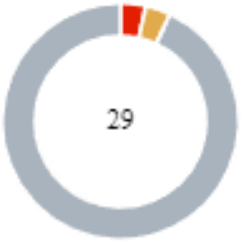
The primary functionality of the CHAPC contract is to manage the minting of NFTs across three distinct packages, each with a predefined token supply limit. The contract supports a presale phase, where only whitelisted addresses are allowed to mint tokens, verified through Merkle proof validation. Additionally, the contract includes mechanisms for airdrop winners, enabling the contract owner to mint NFTs directly to specified addresses.

The contract also integrates a token distribution mechanism, rewarding NFT transfers with tokens 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	1
Medium	1
Minor / Informative	27

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	1	0	0	0
Minor / Informative	27	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	PRE	Potential Reentrance Exploit	Unresolved
	RAMF	Restrictive Airdrop Minting Functionality	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	DCP	Debugging Code Presence	Unresolved
	DDP	Decimal Division Precision	Unresolved
	DMPA	Distribution Mechanism Potential Abuse	Unresolved
	DI	Documentation Inconsistency	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	IVCP	Improper Validity Checks Placement	Unresolved
	IAH	Inefficient Array Handling	Unresolved
	IDT	Inefficient Data Type	Unresolved
	MPC	Merkle Proof Centralization	Unresolved
	MEM	Misleading Error Messages	Unresolved
	MEE	Missing Events Emission	Unresolved

●	PBTMA	Purchased Bones Tracking Mechanism Absence	Unresolved
●	RC	Redundant Check	Unresolved
●	RCS	Redundant Code Segment	Unresolved
●	RRO	Redundant Randomness Operations	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	SMF	Stop Minting Functionality	Unresolved
●	UT	Unnecessary Typecasting	Unresolved
●	UVD	Unnecessary Variable Declaration	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
●	L19	Stable Compiler Version	Unresolved

PRE - Potential Reentrance Exploit

Criticality	Critical
Location	homosapiens.sol#L131,217
Status	Unresolved

Description

The `presaleMint` and `airdropMint` functions include a call to the `_safeMint` function, which can potentially introduce a reentrance vulnerability. During the execution of `_safeMint`, an external call is made, it could re-enter the functions before the state variables are updated. This could allow the mint of more tokens than permitted or repeatedly mint/burn the same token ID in order to make sure tokenIds non-mintable. Specifically, the function updates `totalSupplyOfPackages` only after calling `_safeMint`, making it possible to exploit the state before it is correctly updated.

```
function presaleMint(Cart[] memory cart, bytes32[] calldata
_merkleProof)
    external
    payable
{
    _safeMint(msg.sender, tokenId);
    firstOwners[tokenId] = FirstOwner({
        ownerAddress: msg.sender,
        collectedToken: 0
    });
    totalSupplyOfPackages[packageId] += 1;
    ...
}

function airdropMint() external onlyOwner {
    ...
    _safeMint(msg.sender, tokenId);
    firstOwners[tokenId] = FirstOwner({
        ownerAddress: msg.sender,
        collectedToken: 0
    });
    totalSupplyOfPackages[_packageId] += 1;
    winners[msg.sender].isClaimed = true;
}
```

Recommendation

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

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

RAMF - Restrictive Airdrop Minting Functionality

Criticality	Medium
Location	homosapiens.sol#L217
Status	Unresolved

Description

The `airdropMint` function contains an inconsistency that restricts its intended functionality. This function is designed to allow airdrop winners to mint their allocated NFTs. However, due to the presence of the `onlyOwner` modifier, only the contract owner is permitted to execute this function. Additionally, there is a check within the function that verifies if the caller is listed as an airdrop winner. This creates a conflict because the owner, who is the only entity able to call this function, will generally not be listed as an airdrop winner or should not be listed as an airdrop winner, since this functionality if generally for users that interact with the contract. Consequently, the function cannot be used by the actual airdrop winners to claim their tokens, thereby nullifying the purpose of the airdrop distribution process.

```
function airdropMint() external onlyOwner {
    require(!paused, "the contract is paused");
    require(
        winners[msg.sender].winnerAddress == msg.sender,
        "not in airdrop"
    );
    ...
}
```

Recommendation

To resolve this issue, the `airdropMint` function should be accessible to the actual airdrop winners rather than being restricted to the contract owner. The `onlyOwner` modifier should be removed from this function, allowing any user listed as an airdrop winner to execute it. This adjustment will ensure that the function performs as intended, enabling rightful winners to claim their NFTs without requiring the intervention of the contract owner. This change aligns the function's access control with its intended use, thereby enhancing the contract's operational integrity and usability.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	homosapiens.sol#L401,406,411,419,427,435,441,452,470,480
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 setmaxLimitPerSession(uint16 _newmaxLimitPerSession)
        external
        onlyOwner
    {
        maxLimitPerSession = _newmaxLimitPerSession;
    }

function setTokenDistributing(bool value) external onlyOwner {
        isTokenDistributing = value;
    }

...
```

Recommendation

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

DCP - Debugging Code Presence

Criticality	Minor / Informative
Location	homosapiens.sol#L13,177
Status	Unresolved

Description

The contract includes the `import` statement of `console.sol` of hardhat and utilizes the `console.log` function for debugging purposes. While debugging statements like these are useful during the development and testing phases, they serve no purpose in a deployed contract and result in unnecessary gas costs. The inclusion of debugging code in the final deployed contract can lead to increased transaction fees and overall inefficiency.

```
import "hardhat/console.sol";

console.log(totalPrice);
```

Recommendation

It is recommended to remove all debugging code, including the import statement and any `console.log` statements, from the contract before deployment. This will ensure that the contract is optimized for gas efficiency and does not include any extraneous code that could increase transaction costs.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	homosapiens.sol#L305,306,307
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 creatorTokens = (totalAssignedTokenCount * 60) / 100;
uint256 buyerTokens = (totalAssignedTokenCount * 35) / 100;
uint256 firstOwnerTokens = (totalAssignedTokenCount * 5) / 100;
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

DMPA - Distribution Mechanism Potential Abuse

Criticality	Minor / Informative
Location	homosapiens.sol#L333
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.

DI - Documentation Inconsistency

Criticality	Minor / Informative
Location	homosapiens.sol#L21,23,73,79
Status	Unresolved

Description

The comments at the beginning of the contract indicate that the contract should be unpaused to start the minting process and mention that a batch consists of 500 tokens. Specifically, the comments state: "Unpause contract and start minting process." and "When 1 batch (500 tokens) is minted from each package." However, the actual contract code initializes the `paused` variable to `false`, meaning the contract starts in an unpaused state. Additionally, the `batchSize` constant is set to 50 instead of 500. This discrepancy between the documentation and the actual implementation can lead to confusion.

```
bool public paused = false;

uint256 public constant batchSize = 50;
```

Recommendation

To maintain consistency and avoid confusion, it is recommended to update the comments in the contract to accurately reflect the actual implementation.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	homosapiens.sol#L127
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.

```
s_subscriptionId
```

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.

IVCP - Improper Validity Checks Placement

Criticality	Minor / Informative
Location	homosapiens.sol#L226,366
Status	Unresolved

Description

There are instances where validity checks are placed in functions that handle the later stages of a process, rather than at the initial stages where the data is first received and processed. This misplacement of checks can lead to inefficiencies and potential vulnerabilities. Specifically, in the `requestRandomSeeds` function, the `check require(packageId < totalPackages, "Invalid packageId");` is absent but appears in the `setRandomSeed` function. Similarly, in the `airdropMint` function, the c`heck require(_packageId >= 0 && _packageId < 2, "Invalid package id");` is performed during the minting process but should be included in the `setAirdropWinners` function where the airdrop winners and their package IDs are initially set. Placing these checks in the initial stages ensures that invalid data is caught early, preventing unnecessary execution of subsequent logic and reducing the risk of errors or exploits.

```
function airdropMint() external onlyOwner {
    require(!paused, "the contract is paused");
    require(
        winners[msg.sender].winnerAddress == msg.sender,
        "not in airdrop"
    );
    require(!winners[msg.sender].isClaimed, "already claimed");
    uint256 _packageId = winners[msg.sender].packageId;
    uint256 _mintAmount = 1;
    require(_packageId >= 0 && _packageId < 2, "Invalid package
id");
    ...
}


function setRandomSeed(uint8 packageId, uint256 randomness)
internal {
    require(packageId < totalPackages, "Invalid packageId");
    uint256 batchNumber =
lastRevealedTokenOfPackages[packageId] /
        batchSize;
    randomSeeds[packageId][batchNumber] = randomness;
}
```

Recommendation

To improve the efficiency and security of the contract, it is recommended to move the validity checks to the functions where the data is first processed. By implementing these changes, the contract will be more robust, with early validation of inputs preventing the propagation of invalid data through the contract's logic. This approach enhances both the security and efficiency of the contract operations.

IAH - Inefficient Array Handling

Criticality	Minor / Informative
Location	homosapiens.sol#L
Status	Unresolved

Description

The `tokenDistribution` function includes logic to push the buyer address into the `tokenCollectors` array without checking if the address already exists in the array. This results in potential duplication of entries, leading to inefficiency and increased gas costs. Furthermore, the `tokenCollectors` array is not used anywhere else in the contract, making its maintenance redundant and unnecessary. Maintaining an array of addresses without a clear purpose or utility not only increases storage costs but also complicates the contract without providing any functional benefits.

```
function tokenDistribution(address from, address buyer, uint256
tokenId) private {
        if (lastExecutionTime[buyer] + tokenCollectLimit <
block.timestamp) {
            uint256 totalAssignedTokenCount = 1000;
            uint256 creatorTokens = (totalAssignedTokenCount *
60) / 100;
            uint256 buyerTokens = (totalAssignedTokenCount *
35) / 100;
            uint256 firstOwnerTokens = (totalAssignedTokenCount
* 5) / 100;
            firstOwners[tokenId].collectedToken +=
firstOwnerTokens;
            totalCollectedToken += totalAssignedTokenCount;
            creatorCollectedToken += creatorTokens;
            buyerCollectedToken[buyer] += buyerTokens;
            tokenCollectors.push(buyer);
            ...
```

Recommendation

It is recommended to remove the `tokenCollectors` array and the related logic from the `tokenDistribution` function. Since the array is not utilized elsewhere in the contract, eliminating it will streamline the code, reduce unnecessary gas costs, and improve overall efficiency. If tracking token collectors is deemed necessary in the future, consider implementing a more efficient method, such as using a mapping to avoid duplicate entries and ensure quick look-up times. This change will simplify the contract and enhance its performance.

IDT - Inefficient Data Type

Criticality	Minor / Informative
Location	homosapiens.sol#L53
Status	Unresolved

Description

The `Cart` struct is defined with two fields: `packageId` and `quantity`. The `packageId` field is used to represent the package identifier, which takes values ranging from 0 to 2. Despite this limited range, `packageId` is defined as a `uint256`, which is inefficient. Given the small range of possible values, `packageId` can be more efficiently represented as a `uint8`. Similarly, the `quantity` field is used to represent the number of tokens to be minted, and while it is currently a `uint256`, using a `uint16` for `quantity` would be sufficient and more efficient. These inefficient data type choices increase the storage costs and gas fees unnecessarily.

```
struct Cart {
    uint256 packageId;
    uint256 quantity;
}
```

Recommendation

To optimize the contract and reduce gas costs, it is recommended to update the `Cart` struct to use more appropriate data types. Specifically, change the `packageId` field from uin`t256 to` uint8 `and the` quantity `field from` uint256 `to` uint16`. This adjustment will reduce the amount of storage used and improve the overall efficiency of the contract. Ensuring that data types are chosen appropriately for their intended range not only optimizes performance but also makes the code more readable and easier to understand.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	homosapiens.sol#L144
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
    {
        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 (presaleMintOnlyWhitelisted) {
            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	homosapiens.sol#L327,448
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)

emit TokenCollectStatus(
        from,
        buyer,
        tokenId,
        0,
        "Already collect tokens within the last 10 minutes"
    );
```

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	homosapiens.sol#L401,406,411,419,427,435,441,452,470,480
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 setAirdropWinnersNftPerAddressLimit(uint16
_newNftPerAddressLimit)
        external
        onlyOwner
    {
        airdropWinnersNftPerAddressLimit =
_newNftPerAddressLimit;
    }

function setCallbackGasLimit(uint32 gasLimit) external
onlyOwner {
        callbackGasLimit = gasLimit;
    }

...
```

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.

PBTMA - Purchased Bones Tracking Mechanism Absence

Criticality	Minor / Informative
Location	homosapiens.sol#L207
Status	Unresolved

Description

The contract includes a function `purchaseBone` that allows users to buy bones by specifying the quantity and sending the corresponding Ether. The contract ensures that the total supply of bones does not exceed the maximum limit, which is set to 2000, and that the correct payment is received. However, the contract does not include any mechanism to track ownership or the quantity of bones held by each user. Specifically, the function does not update any state variable or mapping to record the number of bones owned by individual users. As a result, there is no way to determine or verify which users have purchased bones or how many bones each user owns after the purchase. This lack of tracking undermines the utility of the bones, as ownership cannot be validated or managed effectively.

```
function purchaseBone(uint8 quantity) external payable {
    require(
        boneCurrentSupply + quantity <= boneMaxSupply,
        "Exceeds total supply"
    );
    require(msg.value == bonePrice * quantity, "Incorrect
amount sent");
    boneCurrentSupply += quantity;
}
```

Recommendation

To ensure that the ownership and quantity of purchased bones are accurately tracked, it is recommended to implement a mapping that records the number of bones held by each user. The `purchaseBone` function should then be updated to increment the user's balance by the purchased quantity. Implementing this tracking mechanism will provide transparency and allow for effective management of bone ownership, thereby enhancing the functionality and trustworthiness of the contract.

RC - Redundant Check

Criticality	Minor / Informative
Location	homosapiens.sol#L142,284
Status	Unresolved

Description

The `presaleMint` function includes a conditional check `if (presaleMintOnlyWhitelisted)` to verify if the caller is on the whitelist before allowing them to mint tokens during the presale. This check is tied to the boolean variable `presaleMintOnlyWhitelisted`, which is initialized to true and is not changed anywhere in the contract. As a result, the condition will always evaluate to true, making the check redundant. This redundancy complicates the code without providing any additional functionality, as the whitelist verification will always be enforced.

```
function presaleMint(Cart[] memory cart, bytes32[] calldata
_merkleProof)
    external
    payable
{
    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 (presaleMintOnlyWhitelisted) {
        require(
            MerkleProof.verify(_merkleProof,
whitelistMerkleRoot, leaf),
            "Not whitelisted"
        );
    }
    ...
}
```

Furthermore, The `fulfillRandomWords` function, includes a redundant for the total supply of packages. Specifically, the function contains a requirement to verify that the total supply of packages for the requested ID is sufficient compared to the `lastRevealedTokenOfPackages` value. This check is redundant because it is already performed in the `requestRandomSeeds` function before making the VRF request. Moreover, performing this check in the `fulfillRandomWords` function, which is intended to handle the VRF callback, can lead to unintended reverts. Such reverts in a callback function can disrupt the VRF response handling, potentially causing delays or failures in the randomization process.

```
function fulfillRandomWords(
        uint256 _requestId,
        uint256[] memory _randomWords
    ) internal override {
        require(s_requests[_requestId].exists, "request not
found");
        require(

totalSupplyOfPackages[s_requests[_requestId].packageId] >=

lastRevealedTokenOfPackages[s_requests[_requestId].packageId] +
                    batchSize,
            "Insufficient total supply for package"
        );
        s_requests[_requestId].fulfilled = true;
        s_requests[_requestId].randomWords = _randomWords;
        setRandomSeed(s_requests[_requestId].packageId,
_randomWords[0]);
        emit RequestFulfilled(_requestId, _randomWords);
    }
```

Recommendation

To simplify the code and improve its maintainability, it is recommended to remove the redundant check for `presaleMintOnlyWhitelisted`. If there is a need for flexibility in allowing both whitelisted and non-whitelisted presale minting, consider implementing a function to toggle the `presaleMintOnlyWhitelisted` variable. This function should be restricted to the contract owner to control when the presale should be exclusive to whitelisted addresses. By doing so, the contract can adapt to different presale phases, providing both exclusivity and open access as needed. Moreover, it is recommended to remove the redundant supply check from the `fulfillRandomWords` function to prevent unnecessary reverts. The necessary supply checks should be ensured within the `requestRandomSeeds` function before making the VRF request. This will streamline the `fulfillRandomWords` function, ensuring it focuses solely on processing the VRF response and setting the random seed. By doing so, the contract will be more robust and less prone to disruptions during the randomization process, ensuring smoother operations and improved reliability.

RCS - Redundant Code Segment

Criticality	Minor / Informative
Location	homosapiens.sol#L121,344
Status	Unresolved

Description

The contract contains redundant code segments that have been commented out, which do not contribute to the functionality of the contract and may cause confusion for future developers. Specifically, the constructor includes a commented-out `ConfirmedOwner` inheritance, which is already inherited through `VRFConsumerBaseV2Plus`, making it unnecessary. Additionally, there is a commented-out `safeTransferFrom` function with an incomplete implementation and redundant comments.These commented-out sections add clutter to the codebase and can lead to misunderstandings or errors during future maintenance or upgrades. Redundant code segments, even when commented out, reduce the overall readability of the contract.

```
ERC721(_name, _symbol)
VRFConsumerBaseV2Plus(0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1
B)
// ConfirmedOwner(msg.sender)

// function safeTransferFrom(
//     address from,
//     address to,
//     uint256 id
// )
//     public
//     virtual
//     override(ERC721, IERC721)
// {
//     //Fractal Requirements

//     //call the original function that you wanted.
//     super.safeTransferFrom(
//         from,
//         to,
//         id
//     );
//     //do some other stuff
//     test++;
// }
```

Recommendation

It is recommended to remove the redundant and commented-out code segments from the contract. By removing these redundant code segments, the contract will be more streamlined, easier to read, and maintain, ensuring that only the necessary and functional code is present. This will improve the overall quality and professionalism of the contract.

RRO - Redundant Randomness Operations

Criticality	Minor / Informative
Location	homosapiens.sol#L487
Status	Unresolved

Description

The `tokenURI` function includes a call to the `shuffleArray` function, which shuffles an array using a Fisher-Yates algorithm. This operation introduces randomness when generating the token URI for each NFT. However, the randomness for each batch of tokens is already determined and stored in the `revealBatchOfPackages` function. The additional randomness operations in the tokenURI function are redundant and result in unnecessary computations, which complicate the code.

```
function tokenURI(uint256 tokenId) public view override returns
(string memory) {
        uint8 packageId;

        if (tokenId < 5500) {
            packageId = 0;
        } else if (tokenId < 9000) {
            packageId = 1;
        } else if (tokenId < 10000) {
            packageId = 2;
        } else {
            return "out-of-bounds";
        }

        if (tokenId >= (lastRevealedTokenOfPackages[packageId] +
startIdsOfPackages[packageId])) {
            return
string(abi.encodePacked("https://storage.junglebox.io/unrevealed.j
son"));
        } else {
            uint256 starts = (tokenId / batchSize) * batchSize;
            if (tokenId < starts || tokenId >= starts + batchSize)
{
                return "out-of-bounds";
            }
            uint256[batchSize] memory ranges;
            for (uint256 i = 0; i < batchSize; i++) {
                ranges[i] = starts + i;
            }
            uint256 batchNumber = (tokenId -
startIdsOfPackages[packageId]) / batchSize;
            shuffleArray(ranges,
randomSeeds[packageId][batchNumber]);
            uint256 index = tokenId - starts;
            return string(abi.encodePacked(_baseURI(),
ranges[index].toString()));
        }
    }
```

Recommendation

To optimize the contract, it is recommended to eliminate the redundant randomness operations in the `tokenURI` function. Instead, the randomness should be fully handled in the `revealBatchOfPackages` function, ensuring that the token IDs are appropriately randomized when the batch is revealed. By doing so, the `tokenURI` function can simply return the URI based on the pre-determined random order without performing additional shuffling. This adjustment will streamline the code and enhance its readability.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	homosapiens.sol#L401,406,411,419,427,435,441,452,470,480
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 setBaseURI(string memory _newBaseURI) public onlyOwner
{
        baseURI = _newBaseURI;
    }

function setmaxLimitPerSession(uint16 _newmaxLimitPerSession)
        external
        onlyOwner
    {
        maxLimitPerSession = _newmaxLimitPerSession;
    }

...
```

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	homosapiens.sol#L131,217,406
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
{
    require(!paused, "the contract is paused");
    ..
}

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

function pause(bool _state) external onlyOwner {
        paused = _state; //@audit : Something like Stops
Transactions
    }
```

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	homosapiens.sol#L391
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."
    );
    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.

UVD - Unnecessary Variable Declaration

Criticality	Minor / Informative
Location	homosapiens.sol#L175
Status	Unresolved

Description

The `presaleMint` function includes an unnecessary variable declaration: `uint256 packageId = cart[i].packageId;`. This declaration occurs within the loop that processes each item in the cart array. However, the `packageId` variable is used only once within the inner loop, making the declaration redundant. Instead, the code can directly use `cart[i].packageId` without assigning it to a separate variable. Removing this unnecessary variable declaration will simplify the code, reduce potential confusion, and slightly improve gas efficiency.

```
uint256 packageId = cart[i].packageId;
```

Recommendation

It is recommended to eliminate the redundant `packageId` variable declaration within the `presaleMint` function. Instead, use `cart[i].packageId` directly wherever the `packageId` variable is currently used. This will streamline the code, making it cleaner and more efficient. Additionally, removing such redundancies contributes to better readability and maintainability of the contract.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	homosapiens.sol#L64,67,68,72,77,82
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 keyHash =

0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae
bytes32 public airdropWinnersMerkleRoot
uint32 numWords = 1
bool public presaleMintOnlyWhitelisted = true
uint256 public bonePrice = 0.01 ether
uint16 requestConfirmations = 3
```

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	homosapiens.sol#L62,74,91,131,285,286,401,406,411,419,427,435,470,480
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 minutes
mapping(uint256 => RequestStatus) public s_requests
bytes32[] calldata _merkleProof
uint256 _requestId
uint256[] memory _randomWords
string memory _newBaseURI
bool _state
uint16 _newmaxLimitPerSession
uint16 _newNftPerAddressLimit
bytes32 _newMerkleRoot
AirdropWinner[] memory _winners
address[] memory _winners
```

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	homosapiens.sol#L85,90
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.

```
uint16 public airdropWinnersNftPerAddressLimit = 5;

uint16[totalPackages] internal endIdsOfPackages = [4999, 8999,
9999]
```

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	homosapiens.sol#L153,226
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(_packageId >= 0 && _packageId < 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	homosapiens.sol#L462,464,503
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 starts = (tokenId / batchSize) * batchSize
```

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	homosapiens.sol#L507
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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	homosapiens.sol#L2
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 ^0.8.19;
```

Recommendation

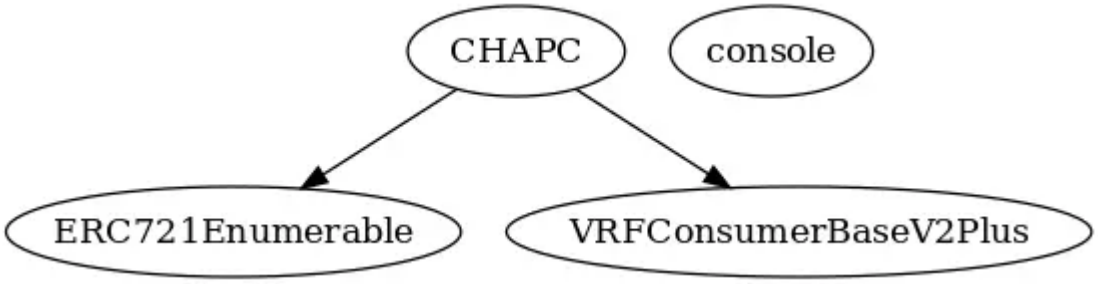
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest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

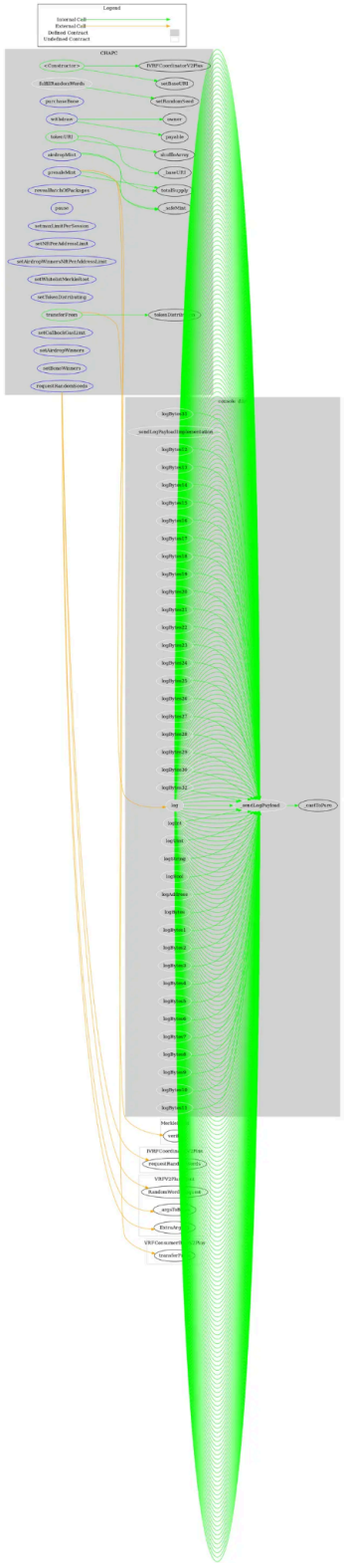
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
CHAPC	Implementation	ERC721Enumerable, VRFConsumerBaseV2Plus		
		Public	✓	ERC721 VRFConsumerBaseV2Plus
	presaleMint	External	Payable	-
	purchaseBone	External	Payable	-
	airdropMint	External	✓	onlyOwner
	requestRandomSeeds	External	✓	onlyOwner
	fulfillRandomWords	Internal	✓	
	tokenDistribution	Private	✓	
	transferFrom	Public	✓	-
	setRandomSeed	Internal	✓	
	revealBatchOfPackages	External	✓	onlyOwner
	_baseURI	Internal		
	setBaseURI	Public	✓	onlyOwner
	pause	External	✓	onlyOwner
	setmaxLimitPerSession	External	✓	onlyOwner
	setNftPerAddressLimit	External	✓	onlyOwner
	setAirdropWinnersNftPerAddressLimit	External	✓	onlyOwner

	setWhitelistMerkleRoot	External	✓	onlyOwner
	setTokenDistributing	External	✓	onlyOwner
	withdraw	External	Payable	onlyOwner
	setCallbackGasLimit	External	✓	onlyOwner
	shuffleArray	Internal		
	setAirdropWinners	External	✓	onlyOwner
	setBoneWinners	External	✓	onlyOwner
	tokenURI	Public		-

Inheritance Graph



Flow Graph



Summary

Crypto Homosapiens contract implements a 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 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