



ONYX LABS

Smart Contract Audit

Prepared For:
YinYangGang

Date:
8/24/22



Table of Contents

Table of Contents	2
Summary	3
Overview	5
Findings	6
YYG-001 Move to ERC721A	7
YYG-002 Missing Withdraw Function	8
YYG-003 Simplification of Whitelist Mint	9
YYG-004 Hard Cap Max Supply	10
YYG-005 _totalMinted() vs totalSupply()	11
YYG-006 Call versus Transfer	12
Disclaimer	13
About	14
Initial Source Code	15
Final Source Code	19




Summary

This report was prepared to summarize the findings of the audit performed for YinYangGang of their ERC-721 minting contract, on August 24th, 2022. The primary objective of the audit was to identify issues and vulnerabilities in the source code. This audit was performed utilizing Manual Review techniques and Static Analysis.

The audit consisted of:

- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Static Analysis utilizing Slither Static Analyzer.
- Unit testing specific functions.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Line-by-line manual review of the entire codebase by auditors.

The following report contains the results of the audit, and recommendations to guard against potential security threats and improve contract functionality.

A  next to a recommendation indicates that the recommendation was implemented prior to the completion of the audit.

We would like to thank the YinYangGang team for their business and cooperation; their openness and communication made this audit a success.



Overview

Project Summary

Project Name	YinYangGang
Description	YinYangGang is an ERC721 NFT mint, consisting of three phases. Phase 1 is a merkle tree controlled paid whitelist mint, phase 2 is a merkle tree controlled free whitelist list, and phase 3 is a paid public mint.
Blockchain	Ethereum
Language	Solidity
Codebase	https://github.com/Xirynx/Yin-Yang-Gang-Smart-Contract
Commit	dd89cff1acdc2774795c9299e738f9e9876ccf64

Audit Summary

Delivery Date	8/24/22
Audit Methodology	Static Analysis, Manual Review, Unit Testing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Resolved
Critical	1	0	0	0	1
High	1	0	0	0	1
Medium	3	0	0	0	3
Low	1	0	0	0	1
Informational	0	0	0	0	0



Audit Scope

ID	File
YYG	MintingContract.sol



Findings

ID	Title	Category	Severity	Status
YYG-001	Move to ERC721A	Optimization	High	Resolved
YYG-002	Missing Withdraw Function	Functional	High	Resolved
YYG-003	Simplification of WL mint	Optimization	Medium	Resolved
YYG-004	Hard Cap Max Supply	Security	Medium	Resolved
YYG-005	_totalMinted() vs totalSupply()	Security	Medium	Resolved
YYG-006	Call vs Transfer	Functional	Low	Resolved



YYG-001 | Move to ERC721A

Category	Severity	Location	Status
Optimization	High	N/A	Resolved

Description

The initial contract was built based on the standard OpenZeppelin ERC721 contract. At this time, it makes sense for most ERC721 minting contracts to inherit ERC721A, developed by Churi Labs. ERC721A offers many optimization advantages, including bulk minting with reduced gas usage.

Recommendation

✓ Replace the inheritance of OZ ERC721 with ERC721A and rework the necessary functions to fit.

Reference: <https://chiru-labs.github.io/ERC721A/#/>



YYG-002 | Missing Withdraw Function

Category	Severity	Location	Status
Functional	Critical	N/A	Resolved

Description

The initial contract was inadvertently missing a function to withdraw funds from the contract. This would result in all collected funds being stuck in the contract.

Recommendation

✓ Insert a function that allows for the withdrawal of funds.

Example:

```
function withdraw() public onlyOwner {
    uint256 balance = address(this).balance;
    (bool callSuccess, ) = payable(msg.sender).call{value: balance}("");
    require(callSuccess, "Call failed");
}
```




YYG-003 | Simplification of Whitelist Mint

Category	Severity	Location	Status
Optimization	Medium	Line 107	Resolved

Description

The function `whitelistMint` can be simplified resulting in further gas optimizations.

Recommendation



Original:

```
function whitelistMint(uint256 amount, bytes memory data, bytes32[] calldata
_merkleProof) external {
    require(currentPhase == Phase.WHITELIST, "Whitelist sale not started");
    require(verifyMultiMint(data, _merkleProof), "Incorrect merkle tree
proof");
    require(amount <= 30, "Mint amount too large for one transaction");
    require(totalSupply() + amount <= maxSupply, "Max supply exceeded");
    (address minter, uint256 maxAmount) = abi.decode(data, (address, uint256));
    require(msg.sender == minter, "Address not whitelisted");
    require(whitelistMintCount[msg.sender] + amount <= maxAmount, "Max mint for
this wallet exceeded");
    whitelistMintCount[msg.sender] += amount;
    _mint(msg.sender, amount);
}
```

Replacement:

```
function whitelistMint(uint256 amount, bytes32[] calldata _merkleProof) external {
    require(currentPhase == Phase.WHITELIST, "Whitelist sale not started");
    bytes memory data = abi.encode(msg.sender, amount);
    require(verifyMultiMint(data, _merkleProof), "Incorrect merkle tree
proof");
    require(totalSupply() + amount <= maxSupply, "Max supply exceeded");
    uint64 auxData = _getAux(msg.sender);
    require(auxData == 0, "Max mint for this phase exceeded");
    _setAux(msg.sender, 1);
    _mint(msg.sender, amount);
}
```



YYG-004 | Hard Cap Max Supply

Category	Severity	Location	Status
Security	Medium	Line 31	Resolved

Description

The contract allows for the maxSupply to be increased unconditionally by the owner using the function setMaxSupply. This means that the supply could be increased at a later date, and additional token minted. This could cause distrust amongst the community, or could lead to malicious actors minting additional tokens if the private keys to the owner wallet were ever compromised.

Recommendation

✓ Add logic to the setMaxSupply function that caps the maxSupply at the intended threshold.

Example

```
function setMaxSupply(uint256 value) public onlyOwner {  
    require(value > 0 && value <= 10000 && totalSupply() <= value, "Invalid  
max supply");  
    maxSupply = value;  
}
```



YYG-005 | `_totalMinted()` vs `totalSupply()`

Category	Severity	Location	Status
Security	Medium	Several	Resolved

Description

The contract inherits `ERC721ABurnable`, allowing for the burning of tokens. When a token is burnt, `totalSupply` is reduced by the number of tokens burnt. The contract uses `totalSupply` to cap the maximum number of minted tokens. `_totalMinted()` will not decrement if tokens are burnt.

Recommendation

✓ Replace `totalSupply()` with `_totalMinted()` where appropriate.

Example

```
require(value > 0 && value <= 10000 && _totalMinted() <= value, "Invalid max supply")
```

```
require(_totalMinted() < maxSupply, "Max supply exceeded");
```

```
require(_totalMinted() + amount <= maxSupply, "Max supply exceeded");
```

```
require(_totalMinted() + amount <= maxSupply, "Max supply exceeded");
```



YYG-006 | Call versus Transfer

Category	Severity	Location	Status
Functional	Low	N/A	Resolved

Description

The preferred method of removing funds from a contract is using a call function due to other methods having hardcoded gas limits that can prevent the transfer of funds from the contract to another smart contract, such as a gnosis safe.

Recommendation

✓ Replace the transfer call used in the withdraw function with call.

Example

```
function withdraw() public onlyOwner {
    uint256 balance = address(this).balance;
    (bool callSuccess, ) = payable(msg.sender).call{value: balance}("");
    require(callSuccess, "Call failed");
}
```



Disclaimer

The audit performed by Onyx Labs is intended to identify function weaknesses, potential security threats, and improve performance. It does not provide a guarantee of contract performance or the absence of any exploitable vulnerabilities. Any alterations to the source code provided could potentially invalidate the analysis provided in this audit report. Onyx Labs is not liable for any losses or damages incurred as a result of contract deployment and the proceeding occurrences.



About

Onyx Labs is a full-service web3 development firm founded in December 2021. Their scope of expertise ranges from smart contract development for a multitude of applications, to web development and audits. By offering best-in-class service, Onyx Labs has helped many successful projects enter the web3 space, or provided assistance with their ongoing development needs.



Initial Source Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Burnable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

contract YinYangGangNFT is ERC721("Yin Yang Gang", "YYG"), ERC721Enumerable,
ERC721Burnable, Ownable {
    uint256 public maxSupply;
    uint256 public mintPrice;
    uint256 public mintIndex;

    bytes32 public merkleRoot;

    enum Phase{
        NONE,
        RAFFLE,
        WHITELIST,
        PUBLIC
    }

    Phase currentPhase = Phase.NONE;

    string internal yin;
    string internal yang;

    mapping(address => uint256) public raffleMintCount;
    mapping(address => uint256) public whitelistMintCount;

    function setMaxSupply(uint256 value) public onlyOwner {
        require(value > 0 && totalSupply() < value, "Invalid max supply");
        maxSupply = value;
    }

    function setMintPrice(uint256 _mintPrice) public onlyOwner {
        mintPrice = _mintPrice;
    }

    function setMerkleRoot(bytes32 _merkleRoot) public onlyOwner {
        merkleRoot = _merkleRoot;
    }
}
```



```
}

function cyclePhases(bytes32 _newMerkleRoot, uint256 _newSupply, uint256
_newMintPrice) external onlyOwner {
    setMintPrice(_newMintPrice);
    setMerkleRoot(_newMerkleRoot);
    setMaxSupply(_newSupply);
    currentPhase = Phase((uint8(currentPhase) + 1) % 4);
}

function stopAllPhases() external onlyOwner {
    currentPhase = Phase.NONE;
}

function setSpecificPhase(Phase _phase) external onlyOwner {
    currentPhase = _phase;
}

function setYin(string calldata _yin) external onlyOwner {
    yin = _yin;
}

function setYang(string calldata _yang) external onlyOwner {
    yang = _yang;
}

function _baseURI() internal view override returns (string memory) {
    if (((block.timestamp + 7200) % 86400) < 36000) {
        return yin;
    } else {
        return yang;
    }
}

function verifySingleMint(address wallet, bytes32[] calldata _merkleProof)
public view returns (bool) {
    bytes32 leaf = keccak256(abi.encodePacked(wallet));
    return MerkleProof.verify(_merkleProof, merkleRoot, leaf);
}

function verifyMultiMint(bytes memory data, bytes32[] calldata _merkleProof)
public view returns (bool) {
    bytes32 leaf = keccak256(abi.encodePacked(data));
    return MerkleProof.verify(_merkleProof, merkleRoot, leaf);
}

function raffleMint(bytes32[] calldata _merkleProof) external payable {
```




```
require(currentPhase == Phase.RAFFLE, "Raffle sale not started");
require(msg.value >= mintPrice, "Insufficient funds");
require(verifySingleMint(msg.sender, _merkleProof), "Incorrect merkle tree
proof");
require(totalSupply() < maxSupply, "Max supply exceeded");
require(raffleMintCount[msg.sender] == 0, "Max mint for this wallet
exceeded");
raffleMintCount[msg.sender] += 1;
_safeMint(msg.sender, ++mintIndex);
}

function whitelistMint(uint256 amount, bytes memory data, bytes32[] calldata
_merkleProof) external {
    require(currentPhase == Phase.WHITELIST, "Whitelist sale not started");
    require(verifyMultiMint(data, _merkleProof), "Incorrect merkle tree
proof");
    (address minter, uint256 maxAmount) = abi.decode(data, (address, uint256));
    require(msg.sender == minter, "Address not whitelisted");
    require(totalSupply() + amount <= maxSupply, "Max supply exceeded");
    require(whitelistMintCount[msg.sender] + amount <= maxAmount, "Max mint for
this wallet exceeded");
    whitelistMintCount[msg.sender] += amount;
    unchecked {
        for (uint256 i = 0; i < amount; i++) {
            _safeMint(msg.sender, ++mintIndex);
        }
    }
}

function publicMint() external payable {
    require(currentPhase == Phase.PUBLIC, "Public sale not started");
    require(tx.origin == msg.sender, "Caller is not origin");
    require(msg.value >= mintPrice, "Insufficient funds");
    require(totalSupply() < maxSupply, "Max supply exceeded");
    _safeMint(msg.sender, ++mintIndex);
}

function adminMint(address to, uint256 amount) external onlyOwner {
    require(totalSupply() + amount <= maxSupply, "Max supply exceeded");
    unchecked {
        for (uint256 i = 0; i < amount; i++) {
            _safeMint(to, ++mintIndex);
        }
    }
}

function _beforeTokenTransfer(address from, address to, uint256 tokenId)
```



```
internal
override(ERC721, ERC721Enumerable)
{
    super._beforeTokenTransfer(from, to, tokenId);
}

function supportsInterface(bytes4 interfaceId)
    public
    view
    override(ERC721, ERC721Enumerable)
    returns (bool)
{
    return super.supportsInterface(interfaceId);
}
```



Final Source Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "erc721a/contracts/ERC721A.sol";
import "erc721a/contracts/extensions/ERC721ABurnable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

contract YinYangGangNFT is ERC721A("Yin Yang Gang", "YYG"), ERC721ABurnable,
Ownable {
    uint256 public maxSupply;
    uint256 public mintPrice;

    bytes32 public merkleRoot;

    enum Phase{
        NONE,
        RAFFLE,
        WHITELIST,
        PUBLIC
    }

    Phase currentPhase = Phase.NONE;

    string internal yin;
    string internal yang;

    function setMaxSupply(uint256 value) public onlyOwner {
        require(value > 0 && value <= 10000 && _totalMinted() <= value, "Invalid
max supply");
        maxSupply = value;
    }

    function setMintPrice(uint256 _mintPrice) public onlyOwner {
        mintPrice = _mintPrice;
    }

    function setMerkleRoot(bytes32 _merkleRoot) public onlyOwner {
        merkleRoot = _merkleRoot;
    }

    function cyclePhases(bytes32 _newMerkleRoot, uint256 _newSupply, uint256
_newMintPrice) external onlyOwner {
        setMintPrice(_newMintPrice);
    }
}
```



```
    setMerkleRoot(_newMerkleRoot);
    setMaxSupply(_newSupply);
    currentPhase = Phase((uint8(currentPhase) + 1) % 4);
}

function stopAllPhases() external onlyOwner {
    currentPhase = Phase.NONE;
}

function setSpecificPhase(Phase _phase) external onlyOwner {
    currentPhase = _phase;
}

function setYin(string calldata _yin) external onlyOwner {
    yin = _yin;
}

function setYang(string calldata _yang) external onlyOwner {
    yang = _yang;
}

function _baseURI() internal view override returns (string memory) {
    if (((block.timestamp + 7200) % 86400) < 36000) {
        return yin;
    } else {
        return yang;
    }
}

function verifySingleMint(address wallet, bytes32[] calldata _merkleProof)
public view returns (bool) {
    bytes32 leaf = keccak256(abi.encodePacked(wallet));
    return MerkleProof.verify(_merkleProof, merkleRoot, leaf);
}

function verifyMultiMint(bytes memory data, bytes32[] calldata _merkleProof)
public view returns (bool) {
    bytes32 leaf = keccak256(abi.encodePacked(data));
    return MerkleProof.verify(_merkleProof, merkleRoot, leaf);
}

function raffleMint(bytes32[] calldata _merkleProof) external payable {
    require(currentPhase == Phase.RAFFLE, "Whitelist sale not started");
    require(msg.value >= mintPrice, "Insufficient funds");
    require(verifySingleMint(msg.sender, _merkleProof), "Incorrect merkle tree
proof");
    require(_totalMinted() < maxSupply, "Max supply exceeded");
}
```



```
uint64 auxData = _getAux(msg.sender);
require(auxData & 1 == 0, "Max mint for this phase exceeded");
_setAux(msg.sender, auxData | 1);
_mint(msg.sender, 1);
}

function whitelistMint(uint256 amount, bytes32[] calldata _merkleProof)
external {
    require(currentPhase == Phase.WHITELIST, "Whitelist sale not started");
    bytes memory data = abi.encode(msg.sender, amount);
    require(verifyMultiMint(data, _merkleProof), "Incorrect merkle tree
proof");
    require(_totalMinted() + amount <= maxSupply, "Max supply exceeded");
    uint64 auxData = _getAux(msg.sender);
    require(auxData & (1 << 1) == 0, "Max mint for this phase exceeded");
    _setAux(msg.sender, auxData | (1 << 1));
    _mint(msg.sender, amount);
}

function publicMint() external payable {
    require(currentPhase == Phase.PUBLIC, "Public sale not started");
    require(tx.origin == msg.sender, "Caller is not origin");
    require(msg.value >= mintPrice, "Insufficient funds");
    require(_totalMinted() < maxSupply, "Max supply exceeded");
    _mint(msg.sender, 1);
}

function adminMint(address to, uint256 amount) external onlyOwner {
    require(_totalMinted() + amount <= maxSupply, "Max supply exceeded");
    require(amount <= 30, "Mint amount too large for one transaction");
    _mint(to, amount);
}

function withdrawFunds(address to) public onlyOwner {
    uint256 balance = address(this).balance;
    (bool callSuccess, ) = payable(to).call{value: balance}("");
    require(callSuccess, "Call failed");
}
}
```