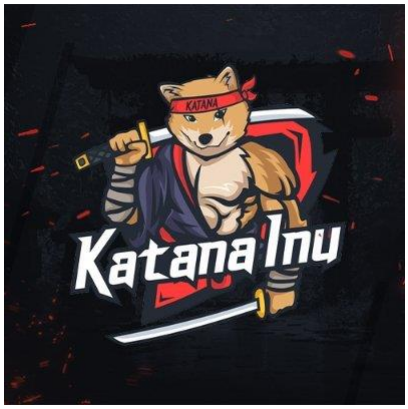
verichains

SECURITY AUDIT OF

KATANA INU NFT SMART CONTRACTS



Public Report

Apr 24, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.
BSC	Binance Smart Chain or BSC is an innovative solution for introducing interoperability and programmability on Binance Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Apr 24, 2023. We would like to thank the Katana Inu for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Katana Inu NFT smart contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team found some vulnerabilities in the application. Katana Inu team has resolved and updated the issue following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Katana Inu NFT smart contracts

Katana Inu is a double-edged revolutionary project that focuses on driving DeFi innovation through its unique game offerings. The Katana Inu platform primarily offers high-end blockchain gaming that combines diverse applications, a real PC Game with NFT equipment, including NFT-staking, charity, and an NFT-marketplace on layer 2. These mechanisms work in unison, creating a self-sustaining token-based ecosystem capable of remitting consistent rewards to all its participants.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of Katana Inu NFT smart contracts.

It was conducted on commit `a7f4ef2195aa9a300703903522d2b776be07ca08` from git repository link: *https://gitlab.com/katana-inu-nft-system/ktn-smc*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
`dd7d1d14b76c202c5f2f63e6b0d5df1b55b1b1252362f5cb5e5c476ef0e43015`	`./Collection.sol`
`84f9129c543b7d4dfc2c75a4d2adb033d6883c3a3fdb20a4442e6492124c5a60`	`./Configurations.sol`
`1570725d7c161db48e98f65d10046abd62ed76e4466a8f051e8907ef2df1c48e`	`./Creator.sol`
`b016cafc49885e9039701ce688509b89e5c1f82caa2703352a359303a11cfd04`	`./Factory.sol`
`76c589bfe5e9a9ec24ee2951066af52ac16e2cb7be4ffb48746a37dfe437bcaa`	`./RoyaltyController.sol`

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Katana Inu NFT smart contracts was written in `Solidity` language. The source code was written based on OpenZeppelin's library.

2.1.1. Collection contract

`Collection` contract is an `ERC721` NFT upgradable contract which extends `ERC721Upgradeable`, `OwnableUpgradeable`, `ERC721RoyaltyUpgradeable`, `AccessControlUpgradeable`, `PausableUpgradeable` and `UUPSUpgradeable` contract. With `OwnableUpgradeable`, by default, Token Owner is contract deployer, but he can transfer ownership to another address at any time. `AccessControlUpgradeable` allows the contract to implement role-based access control mechanisms which let `DEFAULT_ADMIN_ROLE` (contract deployer by default) sets any role for anyone. `ERC721RoyaltyUpgradeable` implements a standardized way to retrieve royalty payment by extends `ERC-2981` - NFT Royalty Standard.

This contract lets `DaapNFTCreator` and `MINTER_ROLE` mint NFTs for users.

2.1.2. DaapNFTCreator contract

`DaapNFTCreator` contract is an upgradable contract which extends `AccessControlUpgradeable`, `PausableUpgradeable` and `OwnableUpgradeable` contract. This contract is the place where users pay tokens to mint NFTs. Users can also redeem `discount` by using a valid signature (generated by the backend) to reduce the cost of minting NFTs.

2.1.3. KatanaNftFactory contract

`KatanaNftFactory` is a factory contract which extends `AccessControl` contract. `AccessControl` allows the contract to implement role-based access control mechanisms which let `DEFAULT_ADMIN_ROLE` (contract deployer by default) sets any role for anyone. By default, the contract sets `DEFAULT_ADMIN_ROLE` and `IMPLEMENTATION_ROLE` for deployer.

This contract lets `IMPLEMENTATION_ROLE` create and configure `Collection`.

2.1.4. RoyaltyController contract

`RoyaltyController` is a royalty splitter contract which extends `AccessControlUpgradeable` contract. `AccessControlUpgradeable` allows the contract to implement role-based access control mechanisms which let `DEFAULT_ADMIN_ROLE` (contract deployer by default) sets any role for anyone. This contract lets `EMERGENCY_ROLE` configure the treasury address and the royalties proportions for each receiver in each collection. Receivers can withdraw their share whenever treasury receive tokens.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Katana Inu NFT smart contracts.

Katana Inu team fixed the code, according to Verichains's draft report in commit `e0e0c5cc1a8cb93cf627fe3752b462553a7d926b`.

#	Issue	Severity	Status
1	Anyone can withdraw all royalty fund	CRITICAL	Fixed
2	`configureRoyaltiesProportions` the second time will make proportions invalid	HIGH	Fixed
3	Users can control `_discount` and make free mint if `signer` is set to address 0	HIGH	Fixed
4	Signature replay	HIGH	Fixed
5	Unsafe `initialize`	HIGH	Fixed

2.2.1. Anyone can withdraw all royalty fund CRITICAL

Affected files:

- RoyaltyController.sol

In `withdraw` function, there is no check to ensure that only wallets with proportions for royalties can withdraw fund. As a result, anyone can withdraw the entire balance of the treasury.

```
function withdraw(
    address tokenAddress,
    uint256 amount,
    address collectionAddress
) external {
    require(
        IERC20(tokenAddress).balanceOf(
            treasuryAddresses[collectionAddress]
        ) -
        s_royalty_configures[collectionAddress][msg.sender]
        .claimedAmount >=
        amount,
        "Royalty Receiver: Invalid withdraw amount"
    );
    IERC20(tokenAddress).transferFrom(
```

```
        treasuryAddresses[collectionAddress],
        msg.sender,
        amount
    );
    s_royalty_configures[collectionAddress][msg.sender]
    .claimedAmount += amount;
    emit Withdraw(msg.sender, amount);
}
```

RECOMMENDATION

Update `s_available` to total amount of royalty fee for collection when claiming and make sure that only wallets with proportions for royalties can withdraw fund.

```
function withdraw(
    address tokenAddress,
    uint256 amount,
    address collectionAddress
) external {
    require(amount > 0, "Royalty Receiver: Can not withdraw zero");

    // remember to update s_available to total amount of royalty fee when claiming for this
to work
    require(
        s_available[collectionAddress] *
s_royalty_configures[collectionAddress][msg.sender].percent / DENOMINATOR -
        s_royalty_configures[collectionAddress][msg.sender].claimedAmount >=
        amount,
        "Royalty Receiver: Invalid withdraw amount"
    );

    s_royalty_configures[collectionAddress][msg.sender].claimedAmount += amount;

    require(IERC20(tokenAddress).transferFrom(
        treasuryAddresses[collectionAddress],
        msg.sender,
        amount
    ), "Royalty Receiver: Transfer failed");

    emit Withdraw(msg.sender, amount);
}
```

UPDATES

- *Apr 21, 2023*: This issue has been acknowledged and fixed by Katana Inu team.

2.2.2. `configureRoyaltiesProportions` the second time will make proportions invalid HIGH

Affected files:

- RoyaltyController.sol

When configuring royalty proportions, if we configure the same collection again and the new `receivers` are different from the old ones, the mapping `s_royalty_configures` will become invalid as it is not removed old records but adding new ones.

```
function configureRoyaltiesProportions(
    address collectionAddress,
    address[] memory receivers,
    uint256[] memory proportions
) external onlyRole(DESIGNER_ROLE) {
    uint256 _sum;
    for (uint i = 0; i < proportions.length; i++) {
        _sum += proportions[i];
    }
    require(
        _sum == DENOMINATOR,
        "RoyaltyReceiver: Invalid proportions for royalties"
    );
    require(
        receivers.length == proportions.length,
        "RoyaltyReceiver: Invalid lenth of reveivers and proportions"
    );
    for (uint256 i = 0; i < receivers.length; i++) {
        s_royalty_configures[collectionAddress][receivers[i]] = PayoutState(proportions[i],
0);
    }
    s_collections.push(collectionAddress);
    s_is_collection[collectionAddress] = true;

    emit ConfigureRoyaltiesProportions(
        collectionAddress,
        receivers,
        proportions
    );
}
```

RECOMMENDATION

Required the contract to only config royalties proportions once. If you want to update config, please make sure to update before anyone withdraw and delete the mapping first.

```
function configureRoyaltiesProportions(
    address collectionAddress,
    address[] memory receivers,
    uint256[] memory proportions
) external onlyRole(DESIGNER_ROLE) {

    require(!s_is_collection[collectionAddress], "RoyaltyReceiver: this collection has been
configured"); // Fix here
```

```
    uint256 _sum;
    for (uint i = 0; i < proportions.length; i++) {
        _sum += proportions[i];
    }
    require(
        _sum == DENOMINATOR,
        "RoyaltyReceiver: Invalid proportions for royalties"
    );
    require(
        receivers.length == proportions.length,
        "RoyaltyReceiver: Invalid lenth of reveivers and proportions"
    );
    for (uint256 i = 0; i < receivers.length; i++) {
        s_royalty_configures[collectionAddress][receivers[i]] = PayoutState(proportions[i],
0);
    }
    s_collections.push(collectionAddress);
    s_is_collection[collectionAddress] = true;

    emit ConfigureRoyaltiesProportions(
        collectionAddress,
        receivers,
        proportions
    );
}
```

UPDATES

- *Apr 17, 2023*: This issue has been acknowledged and fixed by Katana Inu team.

2.2.3. Users can control `_discount` and make free mint if `signer` is set to address 0 HIGH

Affected files:

- Creator.sol

If the contract sets `signer` to address 0 (disable signing requirement for minting), users can control `_discount` and set it to equal to `_amount` to make free mint.

Also, the `_isWhitelistMint` is not being used. Please either remove it or implement its logic.

```
function makeMintingAction(
    ICollection _nftCollection,
    uint256[] memory _nftIndexes,
    uint256 _discount,
    bool _isWhitelistMint,
    Proof memory _proof,
    string memory _callbackData
) external payable notContract {
    require(
```

```
        _nftIndexes.length > 0,
        "Amount of minting NFTs must be greater than 0"
    );
    for (uint256 i = 0; i < _nftIndexes.length; i++) {
        require(
            IConfiguration(nftConfiguration).checkValidMintingAttributes(
                address(_nftCollection),
                _nftIndexes[i]
            ),
            "Invalid NFT Index"
        );
    }
    require(
        verifySignature(
            signer,
            address(_nftCollection),
            _discount,
            _isWhitelistMint,
            _nftIndexes,
            _proof
        ),
        "Invalid Signature"
    );

    uint256 _amount = 0;
    for (uint256 i = 0; i < _nftIndexes.length; i++) {
        _amount += IConfiguration(nftConfiguration).getCollectionPrice(
            address(_nftCollection),
            _nftIndexes[i]
        );
    }
    require(
        payToken.balanceOf(msg.sender) > _amount - _discount,
        "User needs to hold enough token to buy this token"
    );
    payToken.transferFrom(msg.sender, address(this), _amount - _discount);
    _nftCollection.mint(
        _nftIndexes,
        msg.sender,
        _callbackData
    );
    emit MakingMintingAction(_nftIndexes, _discount, msg.sender);
}

function verifySignature(
    address _signer,
    address _nftCollection,
    uint256 _discount,
    bool _isWhitelistMint,
    uint256[] memory _nftIndexes,
    Proof memory _proof
```

```
) private view returns (bool) {
    if (_signer == address(0x0)) {
        return true;
    }
    bytes32 digest = keccak256(
        abi.encode(
            getChainID(),
            msg.sender,
            address(this),
            _nftCollection,
            _discount,
            _isWhitelistMint,
            _nftIndexes,
            _proof.deadline
        )
    );
    address signatory = ecrecover(digest, _proof.v, _proof.r, _proof.s);
    return signatory == _signer && _proof.deadline >= block.timestamp;
}
```

RECOMMENDATION

Do not allow users to control the `_discount` parameter when disabling the signing requirement for minting.

Either remove or implement `_isWhitelistMint` logic.

```
function makeMintingAction(
    ICollection _nftCollection,
    uint256[] memory _nftIndexes,
    uint256 _discount,
    bool _isWhitelistMint,
    Proof memory _proof,
    string memory _callbackData
) external payable notContract {
    require(
        _nftIndexes.length > 0,
        "Amount of minting NFTs must be greater than 0"
    );
    for (uint256 i = 0; i < _nftIndexes.length; i++) {
        require(
            IConfiguration(nftConfiguration).checkValidMintingAttributes(
                address(_nftCollection),
                _nftIndexes[i]
            ),
            "Invalid NFT Index"
        );
    }
    require(
        verifySignature(
            signer,
```

```
            address(_nftCollection),
            _discount,
            _isWhitelistMint,
            _nftIndexes,
            _proof
        ),
        "Invalid Signature"
    );

    uint256 _amount = 0;
    for (uint256 i = 0; i < _nftIndexes.length; i++) {
        _amount += IConfiguration(nftConfiguration).getCollectionPrice(
            address(_nftCollection),
            _nftIndexes[i]
        );
    }

    _discount = _signer == address(0x0) ? 0 : _discount; // FIX here

    require(
        payToken.balanceOf(msg.sender) > _amount - _discount,
        "User needs to hold enough token to buy this token"
    );
    payToken.transferFrom(msg.sender, address(this), _amount - _discount);
    _nftCollection.mint(
        _nftIndexes,
        msg.sender,
        _callbackData
    );
    emit MakingMintingAction(_nftIndexes, _discount, msg.sender);
}

function verifySignature(
    address _signer,
    address _nftCollection,
    uint256 _discount,
    bool _isWhitelistMint,
    uint256[] memory _nftIndexes,
    Proof memory _proof
) private view returns (bool) {
    if (_signer == address(0x0)) {
        return true;
    }
    bytes32 digest = keccak256(
        abi.encode(
            getChainID(),
            msg.sender,
            address(this),
            _nftCollection,
            _discount,
```

```
            _isWhitelistMint,
            _nftIndexes,
            _proof.deadline
        )
    );
    address signatory = ecrecover(digest, _proof.v, _proof.r, _proof.s);
    return signatory == _signer && _proof.deadline >= block.timestamp;
}
```

UPDATES

- *Apr 17, 2023*: This issue has been acknowledged and fixed by Katana Inu team.

2.2.4. Signature replay HIGH

Affected files:

- Creator.sol

In the `makeMintingAction` function, signature is not checked whether it is used or not. User can replay the signature and mint many times.

```
function makeMintingAction(
    ICollection _nftCollection,
    uint256[] memory _nftIndexes,
    uint256 _discount,
    bool _isWhitelistMint,
    Proof memory _proof,
    string memory _callbackData
) external payable notContract {
    require(
        _nftIndexes.length > 0,
        "Amount of minting NFTs must be greater than 0"
    );
    for (uint256 i = 0; i < _nftIndexes.length; i++) {
        require(
            IConfiguration(nftConfiguration).checkValidMintingAttributes(
                address(_nftCollection),
                _nftIndexes[i]
            ),
            "Invalid NFT Index"
        );
    }
    require(
        verifySignature(
            signer,
            address(_nftCollection),
            _discount,
            _isWhitelistMint,
            _nftIndexes,
            _proof
```

```
        ),
        "Invalid Signature"
    );

    uint256 _amount = 0;
    for (uint256 i = 0; i < _nftIndexes.length; i++) {
        _amount += IConfiguration(nftConfiguration).getCollectionPrice(
            address(_nftCollection),
            _nftIndexes[i]
        );
    }

    _discount = _signer == address(0x0) ? 0 : _discount;

    require(
        payToken.balanceOf(msg.sender) > _amount - _discount,
        "User needs to hold enough token to buy this token"
    );
    payToken.transferFrom(msg.sender, address(this), _amount - _discount);
    _nftCollection.mint(
        _nftIndexes,
        msg.sender,
        _callbackData
    );
    emit MakingMintingAction(_nftIndexes, _discount, msg.sender);
}

function verifySignature(
    address _signer,
    address _nftCollection,
    uint256 _discount,
    bool _isWhitelistMint,
    uint256[] memory _nftIndexes,
    Proof memory _proof
) private view returns (bool) {
    if (_signer == address(0x0)) {
        return true;
    }
    bytes32 digest = keccak256(
        abi.encode(
            getChainID(),
            msg.sender,
            address(this),
            _nftCollection,
            _discount,
            _isWhitelistMint,
            _nftIndexes,
            _proof.deadline
        )
    );
```

```
    address signatory = ecrecover(digest, _proof.v, _proof.r, _proof.s);
    return signatory == _signer && _proof.deadline >= block.timestamp;
}
```

RECOMMENDATION

The minting signature should be checked to ensure that it can only be used once.

UPDATES

- *Apr 21, 2023*: This issue has been acknowledged and fixed by Katana Inu team.

2.2.5. Unsafe `initialize` HIGH

Affected files:

- Configurations.sol

The `Configurations` contract is not design to be upgradeable, but it grants admin roles for `msg.sender` while calling `initialize`. Attackers can front run calling this function and steal admin roles right after the contract deployed.

```
contract Configurations is AccessControlUpgradeable {
    constructor(address _nftFactory, address _dappCreator) {
        NFT_FACTORY = _nftFactory;
        DAPP_CREATOR = _dappCreator;
    }

    function initialize() public initializer {
        __AccessControl_init();
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _setupRole(UPGRADER_ROLE, msg.sender);
    }
}
```

RECOMMENDATION

Move granting admin roles logic to the constructor. Also extends `AccessControl` instead of `AccessControlUpgradeable` and remove `UPGRADER_ROLE`.

```
contract Configurations is AccessControl {
    constructor(address _nftFactory, address _dappCreator) {
        NFT_FACTORY = _nftFactory;
        DAPP_CREATOR = _dappCreator;
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }
}
```

UPDATES

- *Apr 21, 2023*: This issue has been acknowledged and fixed by Katana Inu team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 24, 2023*	Public Report	Verichains Lab

Table 2. Report versions history