Security Audit Report for JTeam NFT Contracts

Date: May 16, 2022

Version: 1.2

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.05.16 15:44:03 +08'00'

Contents

Report Manifest

Item	Description
Client	JTeam
Target	JTeam NFT Contracts

Version History

Version	Date	Description
1.0	May 11, 2022	First version
1.1	May 14, 2022	Support the use of ERC721A
1.2	May 16, 2022	Fix compatibility problem & add constraints for mint

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The smart contracts of this project are used to mint and distribute the corresponding NFT tokens, including the following four stages: *angel*, *earlybird*, *presale* and *public-sale*. The first two stages are prepaid, while the latter two require that the buyers need to spend Ethers minting the NFT tokens. As to the assignment of the NFT IDs, though some special numbers are reserved, most of them are not pre-allocated to the buyers. Besides, one address is allowed to mint multiple NFT tokens.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following. Note that the contracts of this project were refactored during the audit, and the prefix of the contracts' names was changed from "Champion" to "EsportsBoy" (e.g., from "ChampionNFT.sol" to "EsportsBoyNFT.sol"). Besides, a new contract named `EsportsBoyNFTA` using the ERC721A template was added to support the USDT-based sales. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project		Commit SHA
JTeam NFT	Version 1	7d232f1cc824dfd76782bfdff8881ac0777d8c6f
	Version 2	1d8b2f403a2eb2003c246690fd1b74b2a4a29ba1

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

[1] https://github.com/JfansSpace/jteam-nft

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

∗ Duplicated item
∗ Verification of the token receiver
∗ Off-chain metadata security

1.3.4 Additional Recommendation

∗ Gas optimization
∗ Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found three potential issues and nine recommendations in the smart contracts, as follows:

- High Risk: 0
- Medium Risk: 1
- Low Risk: 2
- Recommendations: 9

ID	Severity	Description	Category	Status
1	Medium	Improper implementation for the prepaid earlybird mint	DeFi Security	Fixed
2	Low	Being unable to refund the extra payment	DeFi Security	Fixed
3	Low	Inconsistent operation when adding and removing the collaborator	DeFi Security	Fixed
4	-	Remove dead code	Recommendation	Fixed
5	-	Remove unused event	Recommendation	Fixed
6	-	Refactor unnecessary code logic	Recommendation	Fixed
7	-	Add non-zero checks to prevent potential risks	Recommendation	Fixed
8	-	Avoid using require statement in a loop	Recommendation	Fixed
9	-	Fix some typos	Recommendation	Fixed
10	-	Fix wrong log message	Recommendation	Fixed
11	-	Ensure the security of the private key of the owner	Recommendation	-
12	-	Alleviate the concern of centrality problem	Recommendation	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Improper implementation for the prepaid earlybird mint

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The statement at line 212 in the earlyBridMint function requires that the buyers need to spend Ethers minting the NFT tokens. According to the design, however, the earlybird mint is prepaid.

```
206 function earlyBridMint(uint256 quantity, bytes32[] calldata proof) external payable
207     whenNotPaused
208     earlyBirdSaleActive {
209     require(_msgSender() == tx.origin, "No contracts allowed");
210     require(quantity > 0, "quantity must be greater than 0");
211     require(quantity + earlyBridSaleCount <= EARLYBIRD_SUPPLY,"Not enough EARLYBIRD_SUPPLY");
212     require(msg.value >= publicPrice * quantity, "Not enough ETH");
213     require(MerkleProof.verify(proof, earlybirdRoot, keccak256(abi.encodePacked(_msgSender()))),"
        Address is not in earlybird list");
214
```

```
    earlyBridSaleCount += quantity;

    // for (uint i = 0; i < quantity; i++ ) {
    //     _safeMint(_msgSender(), tokenIdTracker_earlyBrid.current());
    //     IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker_earlyBrid.current(), _msgSender());
    //     tokenIdTracker_earlyBrid.increment();
    // }

    for (uint i = 0; i < quantity; i++ ) {
        _safeMint(_msgSender(), getValidTokenId());
        IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker.current(), _msgSender());
        tokenIdTracker.increment();
    }
}
```

Listing 2.1: ChampionNFT.sol

Impact The earlybird mint will not work properly.

Suggestion Remove that require statement.

2.1.2 Being unable to refund the extra payment

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The publicMint function and the presaleMint function are designed to mint NFT tokens for the public-sale stage and the presale stage, respectively. To ensure the payment, they both have a check msg.value >= publicPrice * quantity to guarantee that the received fund should be greater or equal to the value of the tokens being minted. However, if the fund is greater than the value, there does not exist any way to return the extra payment back to the buyer.

```
function publicMint(uint256 quantity) external payable
    whenNotPaused
    publicSaleActive {
    require(_msgSender() == tx.origin, "No contracts allowed");
    require(quantity > 0, "quantity must be greater than 0");
    require(quantity + publicSaleCount <= PUBLI_SUPPLY,"Not enough PUBLI_SUPPLY");
    require(msg.value >= publicPrice * quantity, "Not enough ETH");

    publicSaleCount += quantity;
    for (uint i = 0; i < quantity; i++ ) {
        _safeMint(_msgSender(), getValidTokenId());
        IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker.current(), _msgSender());
        tokenIdTracker.increment();
    }
}

```

```
function presaleMint(uint256 quantity, bytes32[] calldata proof) external payable
  whenNotPaused
  preSaleActive {
  require(_msgSender() == tx.origin, "No contracts allowed");
  require(quantity > 0, "quantity must be greater than 0");
  require(quantity + preSaleCount <= PRE_SUPPLY,"Not enough PRE_SUPPLY");
  require(msg.value >= publicPrice * quantity, "Not enough ETH");
  require(MerkleProof.verify(proof, presaleRoot, keccak256(abi.encodePacked(_msgSender()))),"Address is not in presale list");

  preSaleCount += quantity;
  for (uint i = 0; i < quantity; i++ ) {
    _safeMint(_msgSender(), getValidTokenId());
    IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker.current(), _msgSender());
    tokenIdTracker.increment();
  }
}
```

Listing 2.2: ChampionNFT.sol

Impact The buyer cannot get back the extra payment.

Suggestion N/A

2.1.3 Inconsistent operation when adding and removing the collaborator

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The operation of the delCollaborator function is not consistent with that of the addCollaborator function. Specifically, when adding a collaborator (by invoking the addCollaborator function), it will be recorded in CollaboratorMap and allCollaborators. However, when removing the collaborator (by invoking the delCollaborator function), it will only be removed from allCollaborators.

```
function addCollaborator(address account, uint percentage) public onlyOwner {
   require(account != address(0), "Collaborator cannot be an empty address");
   require(!CollaboratorMap[account].active, "Collaborator already exists");
   require((totalPercentage() + percentage) <= 10000, "totalPercentage will be greater than 10000");

   CollaboratorMap[account] = Collaborator(percentage, true);
   allCollaborators.push(account);
}
```

Listing 2.3: ChampionCollaborator.sol

```
function delCollaborator(address account) public onlyOwner {
   require(account != address(0), "Collaborator cannot be an empty address");
   require(CollaboratorMap[account].active, "Collaborator is not exists");
   delete CollaboratorMap[account];
}
```

Listing 2.4: ChampionCollaborator.sol

Impact N/A

Suggestion Remove the collaborator from `allCollaborators` in the `delCollaborator` function.

2.2 Additional Recommendation

2.2.1 Remove dead code

Status Fixed in Version 2

Introduced by Version 1

Description The else branch from line 68 to line 70 in the `delivery` function would not be executed. As the `setFirstBuy` function is always invoked when minting, `firstBuyMap[tokenId]` will be set to the `account`, which is the `sender` in the `delivery` function (due to the `require` statement in line 61).

```
64function setFirstBuy(uint tokenId, address account) public onlyNFT_Owner {
65    require(IChampionNFT(jt_nft).ownerOf(tokenId) == account, "the tokenId does not belong to account");
66    firstBuyMap[tokenId] = account;
67    emit SetFirstBuy(_msgSender(), tokenId, account);
68}
69
70function delivery(address sender, uint256 tokenId, uint256 value, bytes32 transactionHash, string memory signature) public whenNotPaused onlyValidator{
71    require(IChampionNFT(jt_nft).ownerOf(tokenId) == sender, "the tokenId does not belong to sender");
72    require(!IChampionNFT(jt_nft).isDelivered(tokenId), "the tokenId has been delivered");
73    require(verify(_msgSender(), sender, tokenId, value, transactionHash, signature), "signature verify failed");
74
75    if (firstBuyMap[tokenId] == sender) {
76        require(value >= medalamount_lv1, "not enough medals to delivery - lv1");
77    }
78    else {
79        require(value >= medalamount_lv2, "not enough medals to delivery - lv2");
80    }
81
82    bytes32 hashMsg = keccak256(abi.encodePacked(sender, tokenId, value, transactionHash));
83    bytes32 hashSender = keccak256(abi.encodePacked(_msgSender() , hashMsg));
84
85    require(!transfersSigned[hashSender], "Transfer already signed by this validator");
86    transfersSigned[hashSender] = true;
87
88    uint256 signed = numTransfersSigned[hashMsg];
89    require(!isAlreadyProcessed(signed), "Transfer already processed");
90    // the check above assumes that the case when the value could be overflew will not happen in the addition operation below
91    signed = signed + 1;
92
```

```
    numTransfersSigned[hashMsg] = signed;

    emit SignedForTransferFromForeign(_msgSender(), transactionHash);

    if (signed >= requiredSignatures) {
        // If the bridge contract does not own enough tokens to transfer
        // it will cause funds lock on the home side of the bridge
        numTransfersSigned[hashMsg] = markAsProcessed(signed);
        IChampionNFT(jt_nft).setDelivered(tokenId, true);
        emit Delivery(_msgSender(), sender, tokenId, value, transactionHash);
    }
}
```

Listing 2.5: ChampionNFTBridge.sol

Impact N/A

Suggestion Remove the dead code.

2.2.2 Remove unused event

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The unused event `SetFirstBuy_Validator` can be removed.

```
contract ChampionNFTBridge is BasicBridge, ChampionEIP712Upgradeable {

    /* --- EVENTS --- */
    event SetFirstBuy(address indexed operator, uint tokenId, address account);
    event SetFirstBuy_Validator(address indexed operator, uint tokenId, address account, bytes32 transactionHash);
    event Delivery(address indexed operator, address sender, uint256 tokenId, uint256 value, bytes32 transactionHash);
    event SignedForTransferFromForeign(address indexed signer, bytes32 transactionHash);
    ...
```

Listing 2.6: ChampionNFTBridge.sol

Impact N/A

Suggestion Remove the unused event.

2.2.3 Refactor unnecessary code logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The member named `active` in struct `Collaborator` seems to be not necessary, as it is not used as a switch to enable/disable the collaborator back and forth. As a result, this struct can be refactored, e.g., `mapping(address => uint)`, if the `percentage` is always greater than zero.

```
contract EsportsBoyCollaborator is OwnableUpgradeable {


    struct Collaborator {
        uint percentage;
        bool active;
    }


    address[] private                          allCollaborators;
    mapping(address => Collaborator) private   CollaboratorMap;
    ...
```

Listing 2.7: ChampionCollaborator.sol

Impact N/A

Suggestion Refactor the corresponding code logic.

2.2.4 Add non-zero checks to prevent potential risks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Some important variables can only be set by the owner. However, they still need to be verified to prevent potential risks. Take `publicPrice` as an example, which is used to calculate the received fund (i.e., `publicPrice * quantity`). If `publicPrice` is accidentally set as 0, then the `msg.sender` is able to mint quantity tokens without paying.

```
function setPublicPrice(uint256 amount) external onlyOwner {
    publicPrice = amount;
}
```

Listing 2.8: ChampionNFT.sol

```
function publicMint(uint256 quantity) external payable
    whenNotPaused
    publicSaleActive {
    require(_msgSender() == tx.origin, "No contracts allowed");
    require(quantity > 0, "quantity must be greater than 0");
    require(quantity + publicSaleCount <= PUBLI_SUPPLY,"Not enough PUBLI_SUPPLY");
    require(msg.value >= publicPrice * quantity, "Not enough ETH");

    publicSaleCount += quantity;
    for (uint i = 0; i < quantity; i++ ) {
        _safeMint(_msgSender(), getValidTokenId());
        IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker.current(), _msgSender());
        tokenIdTracker.increment();
    }
}
```

Listing 2.9: ChampionNFT.sol

Furthermore, the following variables, including nft address and amount, should not be zero as well.

```
function __initialize(address _nft, uint amount_lv1, uint amount_lv2) external initializer {
    __BasicBridge_init();
    __ChampionEIP712_init();
    jt_nft = _nft;
    medalamount_lv1 = amount_lv1;
    medalamount_lv2 = amount_lv2;
    deployedAtBlock = block.number;
}

function setNft(address _nft) public onlyOwner{
    jt_nft = _nft;
}
```

Listing 2.10: ChampionNFTBridge.sol

Impact N/A

Suggestion Add the corresponding sanity checks.

2.2.5 Avoid using `require` statement in a loop

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The failure of the `require` statement in a loop may break the execution.

```
function withdraw() external onlyOwner {
    uint256 balance = address(this).balance;
    require(balance > 0, "no balance to withdraw");
    for (uint i = 0; i < allCollaborators.length; i++) {
        Collaborator memory collaborator = CollaboratorMap[allCollaborators[i]];
        if (collaborator.active) {
            (bool sent, bytes memory data) = payable(allCollaborators[i]).call{value: (balance *
                collaborator.percentage) / 10000}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

Listing 2.11: CompionCollaborator.sol

Impact N/A

Suggestion N/A

2.2.6 Fix some typos

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There exist some typos like “earlyBrid” and “angel”.

```
function earlyBridMint(uint256 quantity, bytes32[] calldata proof) external payable
    whenNotPaused
    earlyBirdSaleActive {
    require(_msgSender() == tx.origin, "No contracts allowed");
    require(quantity > 0, "quantity must be greater than 0");
    require(quantity + earlyBridSaleCount <= EARLYBIRD_SUPPLY,"Not enough EARLYBIRD_SUPPLY");
    require(msg.value >= publicPrice * quantity, "Not enough ETH");
    require(MerkleProof.verify(proof, earlybirdRoot, keccak256(abi.encodePacked(_msgSender()))),"Address is not in earlybird list");

    earlyBridSaleCount += quantity;

    // for (uint i = 0; i < quantity; i++ ) {
    //     _safeMint(_msgSender(), tokenIdTracker_earlyBrid.current());
    //     IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker_earlyBrid.current(), _msgSender());
    //     tokenIdTracker_earlyBrid.increment();
    // }

    for (uint i = 0; i < quantity; i++ ) {
        _safeMint(_msgSender(), getValidTokenId());
        IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker.current(), _msgSender());
        tokenIdTracker.increment();
    }
}
```

Listing 2.12: ChampionNFT.sol

```
function angleMint(uint256 quantity, bytes32[] calldata proof) external
    whenNotPaused
    angleSaleActive {
    require(_msgSender() == tx.origin, "No contracts allowed");
    require(quantity > 0, "quantity must be greater than 0");
    require(quantity + angleSaleCount <= ANGEL_SUPPLY, "Not enough ANGEL_SUPPLY");
    require(MerkleProof.verify(proof, angleRoot, keccak256(abi.encodePacked(_msgSender()))),"Address is not in angle list");
    require(angleMintCount[_msgSender()] + quantity <= angleMintLimit[_msgSender()], "the number of caller mint exceeds the upper limit");

    angleMintCount[_msgSender()] += quantity;
    angleSaleCount += quantity;

    for (uint i = 0; i < quantity; i++ ) {
      _safeMint(_msgSender(), tokenIdTracker_angle.current());
      IChampionNFTBridge(bridgeContractAddress).setFirstBuy(tokenIdTracker_angle.current(), _msgSender());
      tokenIdTracker_angle.increment();
    }
}
```

Listing 2.13: ChampionNFT.sol

Impact N/A

Suggestion Fix the typos.

2.2.7 Fix wrong log message

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The log message specified in line 16 is not correct.

```
61 function setRequiredSignatures(uint256 _requiredSignatures) external onlyOwner {
62     require(validatorCount >= _requiredSignatures, "New requiredSignatures should be greater than num of validators");
63     require(_requiredSignatures != 0, "New requiredSignatures should be > than 0");
64     requiredSignatures = _requiredSignatures;
65     emit RequiredSignaturesChanged(_requiredSignatures);
66 }
```

Listing 2.14: BasicBridge.sol

Impact N/A

Suggestion Fix the message.

2.2.8 Ensure the security of the private key of the owner

Description Since the contract owner can set various kinds of important parameters (e.g., the supply and limit of the sales), mint tokens for arbitrary addresses, and withdraw funds from the contract, it is critical to ensure the security of the owners private key. For instance, the multisig wallet can be used for the owner, and the hardware-based private key protection schema (e.g., TEE based solution) can be leveraged.

Impact N/A

Suggestion Ensure the security of the private key of the contract owner.

2.2.9 Alleviate the concern of centrality problem

Description The project has a highly centralized design. Specifically, the owner of the contracts can invoke a number of privilege functions, including minting tokens for arbitrary addresses, and withdrawing funds from the contracts. This is subject to the centrality problem.

Impact N/A

Suggestion Adopt a decentralized mechanism (e.g., DAO) to manage the contracts.