BEOSIN
Blockchain Security

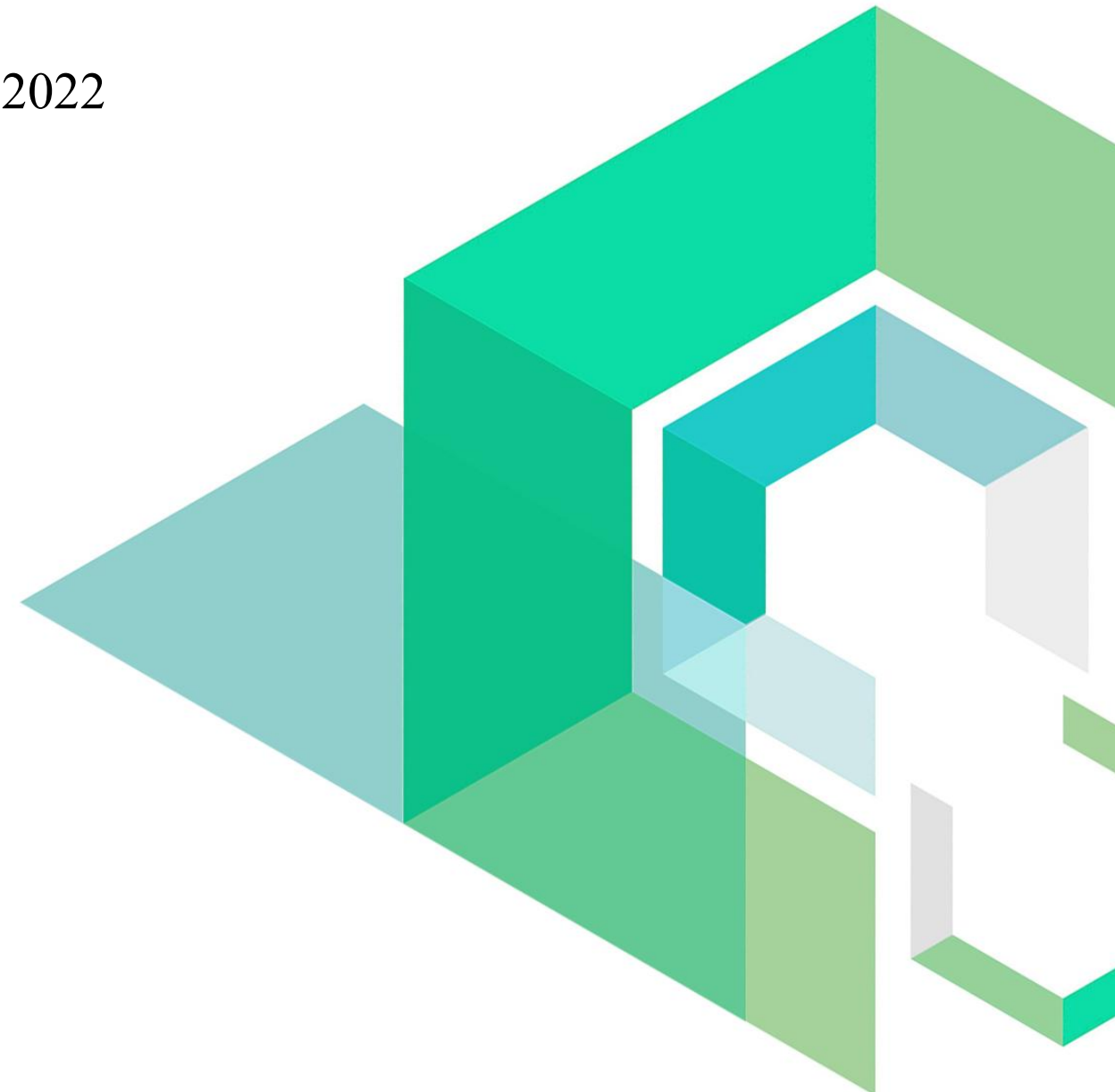
# Hunt Town

Smart Contract Security Audit

V1.0

No. 202212051700

Dec 5th, 2022



# Contents

# Summary of audit results

**After auditing, the Hunt Town project contract was not found to have any risk items.** Specific audit details will be presented in the **Audit Content** section**.**

- **Project Description:**

**1. Business overview**

| | |
|---|---|
| Token name | HUNT Building |
| Token symbol | HUNT_BUILDING |
| Total supply | 18 (mintable) |
| Base uri | https://api.hunt.town/token-metadata/buildings/ |
| Token type | ERC721 |

Table 1 HUNT Building token info

**2. Business overview**

The project is mainly implemented through two contracts, Building and TownHall. The main functions of the Building contract is to mint and burn NFT. These two functions can only be called by the TownHall contract. In the TownHall contract, users can call the *mint* function to lock their 1000 HUNT Tokens. HUNT Building tokens can only be minted through the TownHall contract, and the total amount of minted coins is unlimited. After reaching the one-year locking time, the user can call the *burn* function to burn the NFT and take out the previously locked HUNT Tokens.

# 1 Overview

## 1.1 Project Overview

| Project Name | Hunt Town |
|---|---|
| Platform | Ethereum |
| Contract Address | 0xb09A1410cF4C49F92482F5cd2CbF19b638907193(TownHall) |
| | 0x0c9Bb1ffF512a5B4F01aCA6ad964Ec6D7fC60c96(Building) |

## 1.2 Audit Overview

Audit work duration: Dec 2, 2022 – Dec 5, 2022

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team.

# 2 Audit Content

## 2.1 Detailed Audit of Contract TownHall

(1) mint function

- Description: As shown in the figure below, the *mint* function is used to lock the HUNT Tokens and mint NFT. Minting time is recorded after minting is complete.

```
33      function mint(address to) external {
34          huntToken.safeTransferFrom(msg.sender, address(this), LOCK_UP_AMOUNT);
35          uint256 tokenId = building.safeMint(to);
36
37          buildingMintedAt[tokenId] = block.timestamp;
38
39          emit Mint(to, tokenId, block.timestamp);
40      }
```

Figure 1 Source code of *mint* function

- Related functions: *safeTransferFrom, safemint*

(2) burn function

- Description: This function is used to burn a existing building NFT. The function first checks whether the NFT to be burned has reached the unlock time, if it is reached, it will burn the NFT and refund locked-up HUNT tokens, if not, it will report an error.

```
45      function burn(uint256 tokenId) external {
46          if (block.timestamp < unlockTime(tokenId)) revert TownHall__LockUpPeroidStillLeft();
47
48          // Check approvals and burn the building NFT
49          building.burn(tokenId, msg.sender);
50
51          // Refund locked-up HUNT tokens
52          huntToken.safeTransfer(msg.sender, LOCK_UP_AMOUNT);
53
54          emit Burn(msg.sender, tokenId, block.timestamp);
55      }
```

Figure 2 Source code of *burn* function

- Related functions: *safeTransfer*

(3) mintedAt function

- Description: This function is used to query the minting time of an NFT. The function checks whether NFT exists, if it exists, it returns the minting time, if it does not exist, it reports an error.

```
57      function mintedAt(uint256 tokenId) external view returns (uint256) {
58          if(!building.exists(tokenId)) revert TownHall__InvalidTokenId();
59
60          return buildingMintedAt[tokenId];
61      }
```

Figure 3 Source code of *MintedAt* function

- Related functions: *exists*

(4) unlockTime function

- Description: This contract implements the function of querying the unlocking time of NFT. First check whether an NFT exists, if it exists, return the unlock time of the NFT, if it does not exist, report an error.

```
63      function unlockTime(uint256 tokenId) public view returns (uint256) {
64          if(!building.exists(tokenId)) revert TownHall__InvalidTokenId();
65
66          return buildingMintedAt[tokenId] + LOCK_UP_DURATION;
67      }
```

Figure 4 Source code of *unlockTime* function

- Related functions: *exists*

## 2.2 Detailed Audit of Contract Building

(1) setTownHall function

- Description: This function is used to set the address of the townHall contract. If the address has already been set, an error will be reported. If it is set for the first time, the address of townHall will be set successfully. And this function can only be called by the owner.

```
34      function setTownHall(address _townHall) external onlyOwner {
35          if (townHall != address(0)) revert Building__CannotChangeTownHallAddress();
36
37          townHall = _townHall;
38      }
```

Figure 5 Source code of *setTownHall* function

- Related functions: *setTownHall*

(2) safemint function

- Description: This function can only be called by townHall contract to mint NFT. First take out the current tokenId from _tokenIdCounter, then _tokenIdCounter plus one, and call the internal function *_safeMint* to mint NFT.

```
46      function safeMint(address to) external onlyTownHall returns(uint256 tokenId) {
47          tokenId = _tokenIdCounter.current();
48          _tokenIdCounter.increment();
49
50          _safeMint(to, tokenId);
51      }
```

Figure 6 Source code of *safemint* function

- Related functions: *_safemint*

(3) burn function

- Description: The burn function can only be called by TownHall to burn the NFT. First, the *_isApprovedOrOwner* function will be called to determine whether the msgSender has permission to the NFT to be operated. If so, the *_burn* function will be called to burn it. If not, an error will be reported.

```
60      function burn(uint256 tokenId, address msgSender) external onlyTownHall {
61          if(!_isApprovedOrOwner(msgSender, tokenId)) revert Building__NotOwnerOrApproved();
62
63          _burn(tokenId);
64      }
```

Figure 7 Source code of *burn* function

- Related functions: *_isApprovedOrOwner, _burn*

(4) Other related functions

- Description: As shown in the figure below, several other functions are used to realize the query effect, the *contractURI* function is used to query the contract uri, the *tokenURI* function calls *_baseURI* to query the uri of a certain NFT, *nextId* is used to query the id of the next minted NFT, and the *exists* function is used to check whether an NFT exists, and *unlockTime* calls the unlockTime function in TownHall to return the unlock time of an NFT. It has been checked that there is no safety risk.

```
70      function contractURI() public pure returns (string memory) {
71          return "https://api.hunt.town/token-metadata/buildings.json";
72      }
73
74      function _baseURI() internal view virtual override returns (string memory) {
75          return "https://api.hunt.town/token-metadata/buildings/";
76      }
77
78      function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
79          _requireMinted(tokenId);
80
81          return string(abi.encodePacked(_baseURI(), tokenId.toString(), ".json"));
82      }
83
84      function nextId() external view returns (uint256) {
85          return _tokenIdCounter.current();
86      }
87
88      function exists(uint256 tokenId) external view returns (bool) {
89          return _exists(tokenId);
90      }
91
92      // Utility wrapper function that calls TownHall's unlockTime function
93      function unlockTime(uint256 tokenId) external view returns (uint256) {
94          return ITownHall(townHall).unlockTime(tokenId);
95      }
```

Figure 8 Source code of *contractURI, _baseURI, tokenURI, nextId, exists and unlockTime* functions

- Related functions: *_baseURI, tokenURI, nextId, exists and unlockTime*

# 3 Appendix

## 3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

### 3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

| Likelihood \ Impact | Severe | High | Medium | Low |
|---|---|---|---|---|
| Probable | Critical | High | Medium | Low |
| Possible | High | High | Medium | Low |
| Unlikely | Medium | Medium | Low | Info |
| Rare | Low | Low | Info | Info |

### 3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

### 3.1.3 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

### 3.1.4 Fix Results Status

| Status | Description |
|---|---|
| **Fixed** | The project party fully fixes a vulnerability. |
| **Partially Fixed** | The project party did not fully fix the issue, but only mitigated the issue. |
| **Acknowledged** | The project party confirms and chooses to ignore the issue. |

## 3.2 Audit Categories

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

### 3.2.1 Coding Conventions

Check the code style that does not conform to Solidity code style.

3.2.1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.

3.2.1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.

3.2.1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.

3.2.1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.

3.2.1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.

3.2.1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.

3.2.1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.

3.2.1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.

### 3.2.2 General Vulnerability

Check whether the general vulnerabilities exist in the contract.

3.2.2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.

3.2.2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing BNB.

3.2.2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.

3.2.2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.

3.2.2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.

3.2.2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.

3.2.2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.

3.2.2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.

3.2.2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract.

3.2.2.10 Replay Attack

- Description: Check whether the implement possibility of Replay Attack exists in the contract.

3.2.2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.

.

### 3.2.3 Business Security

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

## 3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in Blockchain.

## 3.4 About BEOSIN

BEOSIN is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions.BEOSIN has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, BEOSIN has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security


**Official Website**

https://www.beosin.com

**Telegram**

https://t.me/+dD8Bnqd133RmNWNl

**Twitter**

https://twitter.com/Beosin_com

**Email**

Contact@beosin.com