Security Audit Report for MonsterToken Contract

Date: December 29, 2025 **Version:** 1.1
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Monster Genesis
Target	MonsterToken Contract

Version History

Version	Date	Description
1.1	December 29, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-12-29 08:52:06+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec 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 successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. 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 target of this audit is the code repository [1] of MonsterToken Contract of Monster Genesis.

The contract `MonsterToken` is an ERC20 token implementing dynamic trade controls, a unidirectional tax mechanism, and strict supply constraints. Specifically, transaction limits are enforced based on real-time reserve depth to prevent whale manipulation, coupled with a mandatory cooldown between trades. Moreover, a lifetime hard cap restricts token supply based on historical cumulative minting, where burning tokens does not restore the quota. Granular whitelists are integrated to allow early buy operations or grant exemptions from cooldown, size cap, and taxes. Taxes are applied to sell or liquidity minting transactions, and are directly transferred to multiple configurable recipients.

Note this audit only focuses on the smart contracts in the following directories/files:

- src/MonsterToken.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. 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. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Code Link and Commit Hash [2]
MonsterToken	`Version 1`	`https://sepolia.etherscan.io/address/0x3670166869cbf89e2b505e430d0a1e68dcf25f7b#code`
	`Version 2`	`5d4ebcb49c11e29a20273f9185c3e1087ad44a22`

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,

[1] `https://github.com/monmondev/monster-token-contract`

[2] The initial version is available on the Sepolia network at `https://sepolia.etherscan.io/address/0x3670166869cbf89e2b505e430d0a1e68dcf25f7b#code`, and the fixed code has been uploaded to GitHub.

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.

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 Security Issues

- Access control
- Permission management
- Whitelist and blacklist mechanisms
- Initialization consistency
- Improper use of the proxy system
- Reentrancy
- Denial of Service (DoS)
- Untrusted external call and control flow
- Exception handling
- Data handling and flow
- Events operation
- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency

∗ Emergency mechanism
∗ Economic and incentive impact

1.3.2 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 [3] and Common Weakness Enumeration [4]. 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

Impact \ Likelihood	*High*	*Low*
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 item will fall into one of the following five categories:

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

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **one** recommendation and **three** notes.

- Medium Risk: 2
- Low Risk: 2
- Recommendation: 1
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	DoS by minimal liquidity provision	Security Issue	Confirmed
2	Medium	Circumvention of trade size restrictions in function `_update()`	Security Issue	Confirmed
3	Low	Lack of total tax rate validation	Security Issue	Fixed
4	Low	Potential circumvention of max supply restriction	Security Issue	Fixed
5	-	Add state change verification in setter functions	Recommendation	Confirmed
6	-	Potential centralization risks	Note	-
7	-	Ensure the initial supply is minted first	Note	-
8	-	Potential imbalanced liquidity provision	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 DoS by minimal liquidity provision

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description During the timing window when the pair is initialized without reserves, an attacker can add liquidity with minimal amounts of tokens. When other users interact with the pool, the calculated `sellSizeCap` and `buySizeCap` can be rounded down to zero. Consequently, the `require` checks on lines 220 and 243 always revert, and all operations from non-whitelist users will be blocked.

```
216         (uint256 reserveToken, ) = _getReservesOrdered();
217         // reserveToken is 0 means adding liquidity
218         if (reserveToken > 0 && !_tradeSizeCapWhitelist.contains(from)) {
219             uint256 sellSizeCap = (reserveToken * sellSizeCapBps) / 10000;
220             require(value <= sellSizeCap, "sell size cap exceeded");
221         }
222     }
```

Listing 2.1: src/MonsterToken.sol

```
        // reserveToken is 0 means adding liquidity
        if (reserveToken > 0 && !_tradeSizeCapWhitelist.contains(to)) {
            uint256 buySizeCap = (reserveToken * buySizeCapBps) / 10000;
            require(value <= buySizeCap, "buy size cap exceeded");
        }
    }
```

Listing 2.2: src/MonsterToken.sol

Impact Malicious actors can cause a permanent DoS (Denial-of-Service) of all future trading and liquidity operations.

Suggestion Initialize the pair with sufficient liquidity immediately after pair creation.

Feedback from the project This scenario will not occur in our implementation. At deployment, there will be no `Monster` token supply available aside from the initial liquidity that will be transferred to the development team as initial LP tokens. Immediately following contract deployment, we will add the initial liquidity to enable trading. Given this controlled deployment process, we prefer to leave this as "Won't Fix."

2.1.2 Circumvention of trade size restrictions in function `_update()`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The function `_update()` enforces a cooldown period per account and a trade size cap, which is computed as a configured proportion of token reserves. However, the cooldown restriction only applies to pair interactions, while direct token transfers have no such limitations. Users could circumvent the cooldown period by transferring tokens to a new address without the `userLastTradeTimestamp` record and trading immediately. By repeatedly performing the above operations, users can deplete nearly all reserves in a transaction, rendering the trade cap limit ineffective.

```
        require(
            block.timestamp - userLastTradeTimestamp[from] >= coldTime ||
                _coldTimeWhitelist.contains(from),
            "sell cold time not elapsed"
        );
        userLastTradeTimestamp[from] = block.timestamp;

        // sell size cap check
        (uint256 reserveToken, ) = _getReservesOrdered();
        // reserveToken is 0 means adding liquidity
        if (reserveToken > 0 && !_tradeSizeCapWhitelist.contains(from)) {
            uint256 sellSizeCap = (reserveToken * sellSizeCapBps) / 10000;
            require(value <= sellSizeCap, "sell size cap exceeded");
        }
    }
```

Listing 2.3: src/MonsterToken.sol

```
        require(
            block.timestamp - userLastTradeTimestamp[to] >= coldTime ||
                _coldTimeWhitelist.contains(to),
            "buy cold time not elapsed"
        );
        userLastTradeTimestamp[to] = block.timestamp;

        // buy size cap check
        (uint256 reserveToken, ) = _getReservesOrdered();
        // reserveToken is 0 means adding liquidity
        if (reserveToken > 0 && !_tradeSizeCapWhitelist.contains(to)) {
            uint256 buySizeCap = (reserveToken * buySizeCapBps) / 10000;
            require(value <= buySizeCap, "buy size cap exceeded");
        }
    }
```

Listing 2.4: src/MonsterToken.sol

Impact The buy and sell size cap restriction can be circumvented.

Suggestion Forbid user transfers during their cooldown period.

Feedback from the project This behavior is acceptable for our use case. Our primary objective is to mitigate excessively frequent trading among typical users. If some users find ways to circumvent this limitation, that is acceptable.

2.1.3 Lack of total tax rate validation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `addTaxSetting()` and `updateTaxSetting()` are intended to configure tax parameters. However, the function lacks a validation that the sum of all `bps` values in the `taxSettings` array does not exceed the maximum basis points (i.e., 10000).

```
    function addTaxSetting(
        address target,
        uint256 bps
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        taxSettings.push(TaxSetting({target: target, bps: bps}));
        emit TaxSettingsAdded(target, bps);
    }

    function updateTaxSetting(
        uint8 index,
        address target,
        uint256 bps
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(index < taxSettings.length, "invalid index");
        taxSettings[index].target = target;
        taxSettings[index].bps = bps;
        emit TaxSettingsUpdated(index, target, bps);
    }
```

Listing 2.5: src/MonsterToken.sol

Impact This could allow the cumulative tax rate to surpass 100%, leading to a potential DoS during transactions.

Suggestion Implement a check in both functions to revert if the new cumulative sum of all tax basis points exceeds 10000.

2.1.4 Potential circumvention of max supply restriction

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `constructor`, there is no check on `initialSupply` to ensure its value is less than or equal to `maxSupply`. If `initialSupply` is larger than `maxSupply`, the initial tokens minted via function `mintInitialSupply()` surpass the restriction on the `maxSupply`. Additionally, this renders the function `mint()` unusable, as the `totalMintedAmount` is already larger than `maxSupply` after initial minting.

```
    constructor(
        string memory _name,
        string memory _symbol,
        uint256 _maxSupply,
        uint256 _initialSupply,
        uint256 _coldTime,
        uint256 _buySizeCapBps,
        uint256 _sellSizeCapBps,
        uint256 _enableBuyTimestamp
    ) ERC20(_name, _symbol) {
        maxSupply = _maxSupply;
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        coldTime = _coldTime;
        buySizeCapBps = _buySizeCapBps;
        sellSizeCapBps = _sellSizeCapBps;
        initialSupply = _initialSupply;
        enableBuyTimestamp = _enableBuyTimestamp;
    }
```

Listing 2.6: src/MonsterToken.sol

```
    function mintInitialSupply() public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(!initialSupplyMinted, "initial supply already minted");
        _mint(msg.sender, initialSupply);
        totalMintedAmount += initialSupply;
        initialSupplyMinted = true;
        emit InitialSupplyMinted(initialSupply);
    }

```

```
function mint(
    address to,
    uint256 amount
) public onlyRole(WHITELISTED_CONTRACT_ROLE) {
    // have to track total supply on our own since when ‘to‘ is zero address, the built-in
        total supply will be decreased,
    // which is not what we want
    require(totalMintedAmount + amount <= maxSupply, "max supply exceeded");
    _mint(to, amount);
    totalMintedAmount += amount;
}
```

Listing 2.7: src/MonsterToken.sol

Impact The restriction of `maxSupply` could be circumvented.

Suggestion Add a check on `initialSupply` against `maxSupply`.

2.2 Recommendation

2.2.1 Add state change verification in setter functions

Status Confirmed

Introduced by `Version 1`

Description Setter functions such as `setPair()` and `toggleBuyWhitelistedUser()` are intended to update critical configurations. However, the current implementation proceeds with state writes and event emission without verifying if the new value differs from the existing one, leading to redundant operations.

```
function setPair(address _pair) public onlyRole(DEFAULT_ADMIN_ROLE) {
    pair = _pair;
    emit PairUpdated(_pair);
}

function toggleBuyWhitelistedUser(
    address account,
    bool isWhitelisted
) public onlyRole(DEFAULT_ADMIN_ROLE) {
    if (isWhitelisted) {
        _buyWhitelist.add(account);
    } else {
        _buyWhitelist.remove(account);
    }
}
```

Listing 2.8: src/MonsterToken.sol

Impact This results in unnecessary gas consumption and event emission when the new value is identical to the current state.

Suggestion Check that the new value differs from the current state before updating storage and emitting an event.

Feedback from the project The project decided not to fix this issue as the associated gas cost is trivial.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `DEFAULT_ADMIN_ROLE`) can conduct sensitive operations, which introduces potential centralization risks. For example, the role `DEFAULT_ADMIN_ROLE` can modify configurations (tax rates, size caps) and authorize the token minting role (i.e., `WHITELISTED_CONTRACT_ROLE`). If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project will use governance to mitigate this.

Clarification from BlockSec On December 27, 2025 [1], the project team renounced the `DEFAULT_ADMIN_ROLE` to ensure existing configurations cannot be modified.

2.3.2 Ensure the initial supply is minted first

Introduced by `Version 1`

Description The functions `mintInitialSupply()` and `mint()` are invoked by privileged roles `DEFAULT_ADMIN_ROLE` and `WHITELISTED_CONTRACT_ROLE`, respectively. However, there is no enforcement in the code to prevent function `mint()` invocation before `mintInitialSupply()` is executed. To ensure proper initialization, the project should ensure that the initial supply is minted before any subsequent minting operations are permitted.

Feedback from the project The project will manually make sure that token minting is done in the correct order.

2.3.3 Potential imbalanced liquidity provision

Introduced by `Version 1`

Description Function `_update()` charges taxes on `Monster` tokens when users add liquidity to the pool. If liquidity is added through the PancakeSwap router, the router calculates the required token amounts based on current pool reserves to maintain the price. However, because taxes are applied to `Monster` tokens during the transfer, the actual `Monster` deposited is less than the router's calculated amount. LP tokens are then minted based on the reduced `Monster` amount, while the excess `USDT` remains in the pool as a donation.

Users should be aware of this behavior when adding liquidity through the PancakeSwap router or frontend interface, as they may receive fewer LP tokens than expected relative to their token contribution.

[1] https://bscscan.com/tx/0xf9e63cf3426cf247f5e3cd3be964fc597635d0ec16524ec016b518320a66a6ed

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS