

Advanced Manual **Smart Contract Audit**

September 13, 2022

Audit requested by



MuscleX

0x22e88b8AbECc7e510c98D55991c626D67cdC52Ea

Table of Contents

1. Audit Summary

- 1.1 Audit scope
- 1.2 Tokenomics
- 1.3 Source Code

2. Disclaimer

3. Global Overview

- 3.1 Informational issues
- 3.2 Low-risk issues
- 3.3 Medium-risk issues
- 3.4 High-risk issues

4. Vulnerabilities Findings

5. Contract Privileges

- 5.1 Maximum Fee Limit Check
- 5.2 Contract Pausability Check
- 5.3 Max Transaction Amount Check
- 5.4 Exclude From Fees Check
- 5.5 Ability to Mint Check
- 5.6 Ability to Blacklist Check
- 5.7 Owner Privileges Check

6. Notes

- 6.1 Notes by Coinsult
- 6.2 Notes by MuscleX

7. Contract Snapshot

8. Website Review

9. Certificate of Proof

Audit Summary

Audit Scope

Project Name	MuscleX
Website	http://musclex.app
Blockchain	Binance Smart Chain
Smart Contract Language	Solidity
Contract Address	0x22e88b8AbECc7e510c98D55991c626D67cdC52Ea
Audit Method	Static Analysis, Manual Review
Date of Audit	13 September 2022

This audit report has been prepared by Coinsult's experts at the request of the client. In this audit, the results of the static analysis and the manual code review will be presented. The purpose of the audit is to see if the functions work as intended, and to identify potential security issues within the smart contract.

The information in this report should be used to understand the risks associated with the smart contract. This report can be used as a guide for the development team on how the contract could possibly be improved by remediating the issues that were identified.

Tokenomics

Rank	Address	Quantity (Token)	Percentage
1	0x7231c5e7ae1916a9e16e0969b29d45a9475efbe2	45,007,920	45.0079%
2	Pinksale: PinkLock V2	33,000,000	33.0000%
3	0x18ea3c3b12131918952c95ddec174a607a49f0cc	21,992,080	21.9921%

Source Code

Coinsult was commissioned by MuscleX to perform an audit based on the following code:

<https://bscscan.com/token/0x22e88b8AbECc7e510c98D55991c626D67cdC52Ea>

Disclaimer

This audit report has been prepared by Coinsult's experts at the request of the client. In this audit, the results of the static analysis and the manual code review will be presented. The purpose of the audit is to see if the functions work as intended, and to identify potential security issues within the smart contract.

The information in this report should be used to understand the risks associated with the smart contract. This report can be used as a guide for the development team on how the contract could possibly be improved by remediating the issues that were identified.

Coinsult is not responsible if a project turns out to be a scam, rug-pull or honeypot. We only provide a detailed analysis for your own research.

Coinsult is not responsible for any financial losses. Nothing in this contract audit is financial advice, please do your own research.

The information provided in this audit is for informational purposes only and should not be considered investment advice. Coinsult does not endorse, recommend, support or suggest to invest in any project.

Coinsult can not be held responsible for when a project turns out to be a rug-pull, honeypot or scam.

Global Overview

Manual Code Review

In this audit report we will highlight the following issues:

Vulnerability Level	Total	Pending	Acknowledged	Resolved
● Informational	0	0	0	0
● Low-Risk	7	7	0	0
● Medium-Risk	1	0	0	1
● High-Risk	1	0	0	1

Privilege Overview

Coinsult checked the following privileges:

Contract Privilege	Description
Owner can mint?	● Owner cannot mint new tokens
Owner can blacklist?	● Owner cannot blacklist addresses
Owner can set fees > 25%?	● Owner cannot set the sell fee to 25% or higher
Owner can exclude from fees?	● Owner can exclude from fees
Owner can pause trading?	● Owner cannot pause the contract
Owner can set Max TX amount?	● Owner cannot set max transaction amount

More owner privileges are listed later in the report.

● **Low-Risk:** Could be fixed, will not bring problems.

No need to use if statement to set the router

```
if (block.chainid == 56) {
    dexRouter = IRouter02(0x10ED43C718714eb63d5aA57B78B54704E256024E);
} else if (block.chainid == 97) {
    dexRouter = IRouter02(0xD99D1c33F9fC3444f8101754aBC46c52416550D1);
} else if (block.chainid == 1 || block.chainid == 4) {
    dexRouter = IRouter02(0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D);
} else if (block.chainid == 43114) {
    dexRouter = IRouter02(0x60aE616a2155Ee3d9A68541Ba4544862310933d4);
} else if (block.chainid == 250) {
    dexRouter = IRouter02(0xF491e7B69E4244ad4002BC14e878a34207E38c29);
} else {
    revert();
}
```

Recommendation

Hardcode the router to the BSC

● **Low-Risk:** Could be fixed, will not bring problems.

No zero address validation for some functions

Detect missing zero address validation.

```
// Function to set an operator to allow someone other the deployer to create things such as launchpads.
// Only callable by original deployer.
function setOperator(address newOperator) external {
    require(msg.sender == originalDeployer, "Can only be called by original deployer.");
    address oldOperator = operator;
    if (oldOperator != address(0)) {
        _liquidityHolders[oldOperator] = false;
        setExcludedFromFees(oldOperator, false);
    }
    operator = newOperator;
    _liquidityHolders[newOperator] = true;
    setExcludedFromFees(newOperator, true);
}
```

Recommendation

Check that the new address is not zero.

Exploit scenario

```
contract C {

    modifier onlyAdmin {
        if (msg.sender != owner) throw;
        _;
    }

    function updateOwner(address newOwner) onlyAdmin external {
        owner = newOwner;
    }
}
```

Bob calls updateOwner without specifying the newOwner, so Bob loses ownership of the contract.

● **Low-Risk:** Could be fixed, will not bring problems.

Functions that send Ether to arbitrary destinations

Unprotected call to a function sending Ether to an arbitrary address.

```
if (ratios.buyback > 0) {
    (success,) = _taxWallets.buyback.call{value: buybackBalance, gas: 35000}("");
}
if (ratios.lottery > 0) {
    (success,) = _taxWallets.lottery.call{value: stakingBalance, gas: 35000}("");
}
if (ratios.marketing > 0) {
    (success,) = _taxWallets.marketing.call{value: marketingBalance, gas: 35000}("");
}
```

Recommendation

Ensure that an arbitrary user cannot withdraw unauthorized funds.

Exploit scenario

```
contract ArbitrarySend{
    address destination;
    function setDestination(){
        destination = msg.sender;
    }

    function withdraw() public{
        destination.transfer(this.balance);
    }
}
```

Bob calls setDestination and withdraw. As a result he withdraws the contract's balance.

● **Low-Risk:** Could be fixed, will not bring problems.

Divide before multiply

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision.

```
-toLiquify = ((contractTokenBalance * ratios.liquidity) / ratios.totalSwap) / 2 (#506)
-liquidityBalance = (amtBalance * toLiquify) / swapAmt (#524)
```

Recommendation

Consider ordering multiplication before division.

Exploit scenario

```
contract A {
    function f(uint n) public {
        coins = (oldSupply / n) * interest;
    }
}
```

If n is greater than $oldSupply$, $coins$ will be zero. For example, with $oldSupply = 5$; $n = 10$, $interest = 2$, $coins$ will be zero. If $(oldSupply * interest / n)$ was used, $coins$ would have been 1. In general, it's usually a good idea to re-arrange arithmetic to perform multiplication before division, unless the limit of a smaller type makes this dangerous.

● **Low-Risk:** Could be fixed, will not bring problems.

Missing events arithmetic

Detect missing events for critical arithmetic parameters.

```
function setSwapSettings(uint256 thresholdPercent, uint256 thresholdDivisor, uint256 amountPercent, uint256 amountDivisor) {
    swapThreshold = (_tTotal * thresholdPercent) / thresholdDivisor;
    swapAmount = (_tTotal * amountPercent) / amountDivisor;
    require(swapThreshold <= swapAmount, "Threshold cannot be above amount.");
    require(swapAmount = _tTotal / 1_000_000, "Cannot be lower than 0.00001% of total supply.");
    require(swapThreshold >= _tTotal / 1_000_000, "Cannot be lower than 0.00001% of total supply.");
}
```

Recommendation

Emit an event for critical parameter changes.

Exploit scenario

```
contract C {

    modifier onlyAdmin {
        if (msg.sender != owner) throw;
        _;
    }

    function updateOwner(address newOwner) onlyAdmin external {
        owner = newOwner;
    }
}
```

updateOwner() has no event, so it is difficult to track off-chain changes in the buy price.

● **Low-Risk:** Could be fixed, will not bring problems.

Boolean equality

Detects the comparison to boolean constants.

```
function setLpPair(address pair, bool enabled) external onlyOwner {
    if (enabled == false) {
        lpPairs[pair] = false;
        antiSnipe.setLpPair(pair, false);
    } else {
        if (timeSinceLastPair != 0) {
            require(block.timestamp - timeSinceLastPair > 3 days, "3 Day cooldown!");
        }
        lpPairs[pair] = true;
        timeSinceLastPair = block.timestamp;
        antiSnipe.setLpPair(pair, true);
    }
}
```

Recommendation

Remove the equality to the boolean constant.

Exploit scenario

```
contract A {
    function f(bool x) public {
        // ...
        if (x == true) { // bad!
            // ...
        }
        // ...
    }
}
```

Boolean constants can be used directly and do not need to be compare to true or false.

● **Low-Risk:** Could be fixed, will not bring problems.

Costly operations inside a loop

Costly operations inside a loop might waste gas, so optimizations are justified.

```
function multiSendTokens(address[] memory accounts, uint256[] memory amounts) external {
    require(accounts.length == amounts.length, "Lengths do not match.");
    for (uint8 i = 0; i = amount);
        _transfer(msg.sender, accounts[i], amount);
    }
}
```

Recommendation

Use a local variable to hold the loop computation result.

Exploit scenario

```
contract CostlyOperationsInLoop{

    function bad() external{
        for (uint i=0; i < loop_count; i++){
            state_variable++;
        }
    }

    function good() external{
        uint local_variable = state_variable;
        for (uint i=0; i < loop_count; i++){
            local_variable++;
        }
        state_variable = local_variable;
    }
}
```

Incrementing `state_variable` in a loop incurs a lot of gas because of expensive `SSTOREs`, which might lead to an out-of-gas.

● **Medium-Risk:** Should be fixed, could bring problems.

Unverified antisnipe contract interactions

```
function setInitializer(address initializer) external onlyOwner {
    require(!_hasLiqBeenAdded);
    require(initializer != address(this), "Can't be self.");
    antiSnipe = AntiSnipe(initializer);
}
```

Recommendation

Owner can change the address of the antisniper contract. This contract is unverified and not audited by Coinsult. Current antisniper contract:

<https://bscscan.com/address/0x15945C18175E7287C251414a9578AEf5F0D4aeaF#code>

✓ Contract ownership is held by 'Trynos' (SAFU developer). He is the only one who is able to change the antiSniper contract before the liquidity is added. Therefore this issue is resolved.

● **High-Risk:** Must be fixed, will bring problems.

Assigning the contract address(this) to antiSnipe

```
function _checkLiquidityAdd(address from, address to) private {
    require(!_hasLiqBeenAdded, "Liquidity already added and marked.");
    if (!_hasLimits(from, to) && to == lpPair) {
        _liquidityHolders[from] = true;
        _hasLiqBeenAdded = true;
        if(address(antiSnipe) == address(0)){
            antiSnipe = AntiSnipe(address(this));
        }
        contractSwapEnabled = true;
        emit ContractSwapEnabledUpdated(true);
    }
}
```

Recommendation

Assigning the contract address(this) to the antiSnipe will incur in errors since there is no interface within this contract for antiSnipe.

✅ Contract ownership is held by 'Trynos' (SAFU developer). He set the antiSnipe contract to a correct CA, so this if statement will not hold. Therefore this issue is resolved.

Contract Privileges

Maximum Fee Limit Check


Coinsult tests if the owner of the smart contract can set the transfer, buy or sell fee to 25% or more. It is bad practice to set the fees to 25% or more, because owners can prevent healthy trading or even stop trading when the fees are set too high.

Type of fee	Description
Transfer fee	● Owner cannot set the transfer fee to 25% or higher
Buy fee	● Owner cannot set the buy fee to 25% or higher
Sell fee	● Owner cannot set the sell fee to 25% or higher

Type of fee	Description
Max transfer fee	25%
Max buy fee	25%
Max sell fee	25%


Contract Pausability Check

Coinsult tests if the owner of the smart contract has the ability to pause the contract. If this is the case, users can no longer interact with the smart contract; users can no longer trade the token.

Privilege Check	Description
Can owner pause the contract?	 Owner cannot pause the contract


Max Transaction Amount Check

Coinsult tests if the owner of the smart contract can set the maximum amount of a transaction. If the transaction exceeds this limit, the transaction will revert. Owners could prevent normal transactions to take place if they abuse this function.

Privilege Check	Description
Can owner set max tx amount?	 Owner cannot set max transaction amount

Exclude From Fees Check

Coinsult tests if the owner of the smart contract can exclude addresses from paying tax fees. If the owner of the smart contract can exclude from fees, they could set high tax fees and exclude themselves from fees and benefit from 0% trading fees. However, some smart contracts require this function to exclude routers, dex, cex or other contracts / wallets from fees.


Privilege Check	Description
Can owner exclude from fees?	 Owner can exclude from fees

Ability To Mint Check

Coinsult tests if the owner of the smart contract can mint new tokens. If the contract contains a mint function, we refer to the token's total supply as non-fixed, allowing the token owner to "mint" more tokens whenever they want.

A mint function in the smart contract allows minting tokens at a later stage. A method to disable minting can also be added to stop the minting process irreversibly.


Minting tokens is done by sending a transaction that creates new tokens inside of the token smart contract. With the help of the smart contract function, an unlimited number of tokens can be created without spending additional energy or money.

Privilege Check	Description
Can owner mint?	 Owner cannot mint new tokens

Ability To Blacklist Check

Coinsult tests if the owner of the smart contract can blacklist accounts from interacting with the smart contract. Blacklisting methods allow the contract owner to enter wallet addresses which are not allowed to interact with the smart contract.

This method can be abused by token owners to prevent certain / all holders from trading the token. However, blacklists might be good for tokens that want to rule out certain addresses from interacting with a smart contract.

Privilege Check	Description
Can owner blacklist?	 Owner cannot blacklist addresses

Other Owner Privileges Check

Coinconsult lists all important contract methods which the owner can interact with.

✔ No other important owner privileges to mention.

Notes

Notes by MuscleX

No notes provided by the team.

Notes by Coinconsult

 No notes provided by Coinconsult

Contract Snapshot

This is how the constructor of the contract looked at the time of auditing the smart contract.

```
contract MuscleX is IERC20 {
    mapping (address => uint256) private _tOwned;
    mapping (address => bool) lpPairs;
    uint256 private timeSinceLastPair = 0;
    mapping (address => mapping (address => uint256)) private _allowances;

    mapping (address => bool) private _isExcludedFromFees;
    mapping (address => bool) private _isExcludedFromProtection;
    mapping (address => bool) private presaleAddresses;
    bool private allowedPresaleExclusion = true;
    mapping (address => bool) private _liquidityHolders;

    uint256 constant private startingSupply = 100_000_000;

    string constant private _name = "MuscleX";
    string constant private _symbol = "M-X";
    uint8 constant private _decimals = 18;
    uint256 constant private _tTotal = startingSupply * 10**_decimals;

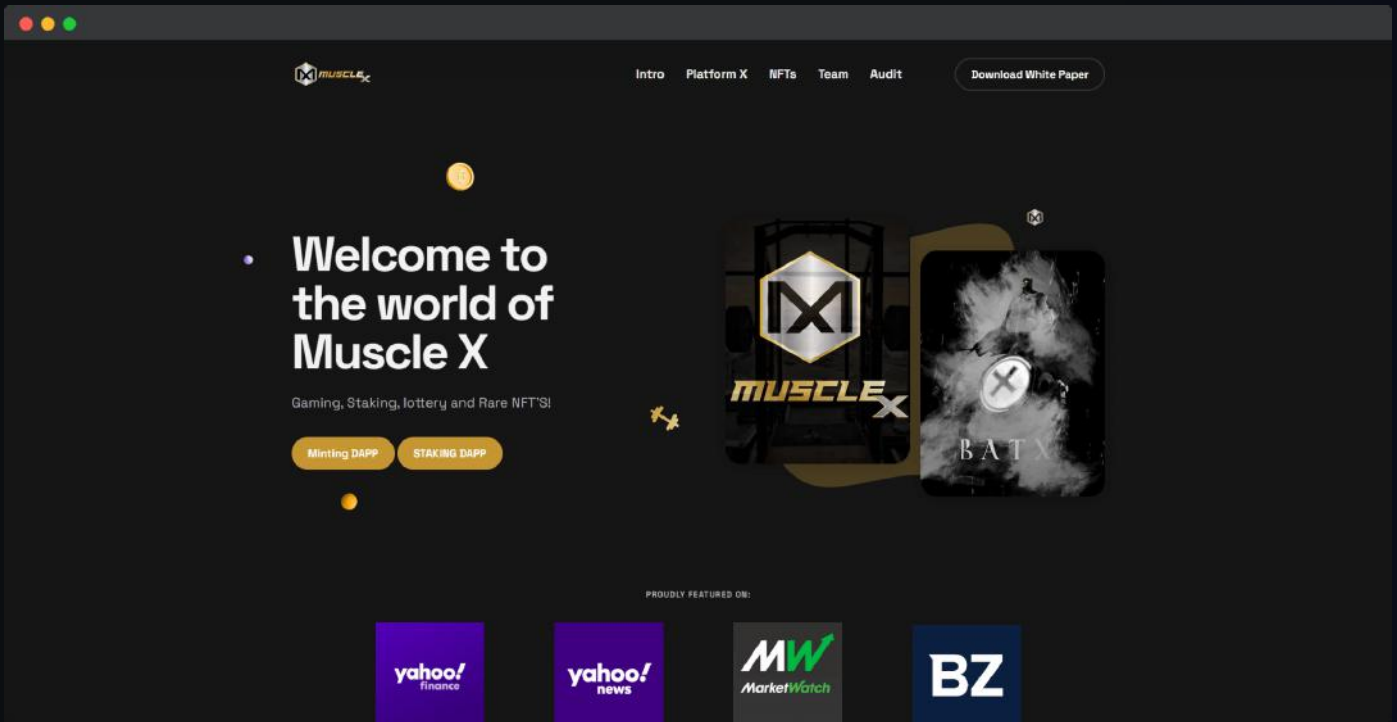
    struct Fees {
        uint16 buyFee;
        uint16 sellFee;
        uint16 transferFee;
    }

    struct Ratios {
        uint16 liquidity;
        uint16 marketing;
        uint16 buyback;
        uint16 lottery;
        uint16 tokens;
        uint16 totalSwap;
    }

    Fees public _taxRates = Fees({
        buyFee: 1100,
        sellFee: 1300,
```


Website Review

Coinsult checks the website completely manually and looks for visual, technical and textual errors. We also look at the security, speed and accessibility of the website. In short, a complete check to see if the website meets the current standard of the web development industry.



Type of check	Description
Mobile friendly?	● The website is mobile friendly
Contains jQuery errors?	● The website does not contain jQuery errors
Is SSL secured?	● The website is SSL secured
Contains spelling errors?	● The website does not contain spelling errors

Certificate of Proof

● Not KYC verified by Coinsult

MuscleX

Audited by Coinsult.net



Date: 13 September 2022

✓ Advanced Manual Smart Contract Audit

End of report
Smart Contract Audit

Request your smart contract audit / KYC

t.me/coinsult_tg