Cyberscope

Audit Report

MagnumBit

May 2024

Network	BSC_TESTNET
Address	0x7b4aba2cdeb6adf036f0ac940d068daa386c5a82
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Unresolved
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	MFC	Misconfigured Fee Calculation	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	RCU	Redundant Check Usage	Unresolved
●	RFM	Redundant Fee Mechanism	Unresolved
●	RFU	Redundant Functionalities Usage	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L16	Validate Variable Setters	Unresolved

Table of Contents

Review

Contract Name	MetaForge
Compiler Version	v0.8.19+commit.7dd6d404
Optimization	200 runs
Explorer	https://testnet.bscscan.com/address/0x7b4aba2cdeb6adf036f0ac940d068daa386c5a82
Address	0x7b4aba2cdeb6adf036f0ac940d068daa386c5a82
Network	BSC_TESTNET
Symbol	MFORGE
Decimals	18
Total Supply	109,996,986.206
Badge Eligibility	Must Fix Criticals

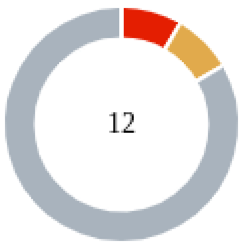
Audit Updates

Initial Audit	27 Apr 2024
Corrected Phase 2	08 May 2024

Source Files

Filename	SHA256
contracts/Mforge.sol	8503763d80dd1fa34670fbc28759516365f1189770d94088141fa829aa093cdb

Findings Breakdown



🟥	Critical	1
🟧	Medium	1
●	Minor / Informative	10

	Severity	Unresolved	Acknowledged	Resolved	Other
🟥	Critical	1	0	0	0
🟧	Medium	1	0	0	0
●	Minor / Informative	10	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	contracts/Mforge.sol#L203,310,332
Status	Unresolved

Description

The contract owner has the authority to stop the sales for all users excluding the owner. If any of the `adminWallet` , `boostLiquidityWallet` , `lpProviderWallet` , `rewardsPoolWallet` addresses is set to zero via the `setAddresses` function, the contract will prevent the successful execution of the transfers.

```
function setAddresses(
    address _adminWallet,
    address _boostLiquidityWallet,
    address _lpProviderWallet,
    address _rewardsPoolWallet
) external onlyOwner {
    adminWallet = _adminWallet;
    boostLiquidityWallet = _boostLiquidityWallet;
    lpProviderWallet = _lpProviderWallet;
    rewardsPoolWallet = _rewardsPoolWallet;
    emit AddressesSet(
        _adminWallet,
        _boostLiquidityWallet,
        _lpProviderWallet,
        _rewardsPoolWallet
    );
}
...
super._transfer(from, adminWallet, adminFee);
_burn(from, burnFee);
super._transfer(from, boostLiquidityWallet, boostLpFee);
super._transfer(from, lpProviderWallet, lpProviderFee);
...
super._transfer(from, adminWallet, adminFee);
_burn(from, burnFee);
super._transfer(from, rewardsPoolWallet, rewardPoolFee);
super._transfer(from, lpProviderWallet, lpProviderFee);
```

Recommendation

The contract should prevent the set of the addresses to the zero address. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

MFC - Misconfigured Fee Calculation

Criticality	Medium
Location	contracts/Mforge.sol#L56,140,162,297,320
Status	Unresolved

Description

The contract is designed to allow for up to 10% fees on both marketing and liquidity variables as per the function parameters in `setBuyFees` and `setSellFees` . These functions are limited to accept values up to 10 for each fee type, with the intention implied through comments to enable a total potential fee of 20% (10% from marketing and 10% from liquidity). However, given that `feeDenominator` is set to `1000` , applying a fee value of 10 only results in a 1% fee per variable, thus the total possible applied fees amount to 2% instead of the intended 20%. This discrepancy stems from a misunderstanding in the relationship between the fee values set in the functions and the actual percentage applied based on the denominator value.

```
   uint16 public immutable feeDenominator = 1000;

    function setBuyFees(

        uint8 marketingFee,

        uint8 liquidityFee

    ) external onlyOwner {
        require(marketingFee <= 10, "Marketing fees cannot exceeds 10%");
        require(liquidityFee <= 10, "Liquidity fees cannot exceeds 10%");

        buyMarketingFee = marketingFee;
        buyLiquidityFee = liquidityFee;

        buyTotalFees = buyMarketingFee + buyLiquidityFee;
        ...
    }

    function setSellFees(
        uint8 marketingFee,
        uint8 liquidityFee
    ) external onlyOwner {
        require(marketingFee <= 10, "Marketing fees cannot exceeds 10%");
        require(liquidityFee <= 10, "Liquidity fees cannot exceeds 10%");

        sellMarketingFee = marketingFee;
        sellLiquidityFee = liquidityFee;
        ...
    }
uint256 newTokensForMarketing = (amount * sellMarketingFee) /
    feeDenominator;
uint256 newTokensForLiquidity = (amount * sellLiquidityFee) /
    feeDenominator;
...
uint256 newTokensForMarketing = (amount * buyMarketingFee) /
    feeDenominator;
uint256 newTokensForLiquidity = (amount * buyLiquidityFee) /
    feeDenominator;
```

Recommendation

It is recommended to reconsider the calculation and the values of the fee variables and limits. If the intent is to allow fees of up to 10% on each variable, then the corresponding code should use a `feeDenominator` of `100` or adjust the fee limits to 100 in the respective functions. This change will align the intended fee structure with the actual calculations performed by the contract, ensuring that the fee settings are transparent and meet the intended economic model. Additionally, enhancing the comments or documentation to clearly describe the fee calculation methodology will prevent future misinterpretations.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L305,327
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 adminFee = (fees * 500) / feeDenominator;
uint256 burnFee = (fees * 200) / feeDenominator;
uint256 boostLpFee = (fees * 100) / feeDenominator;
uint256 lpProviderFee = (fees * 200) / feeDenominator;
...
uint256 adminFee = (fees * 600 ) / feeDenominator;
uint256 burnFee = (fees * 100 ) / feeDenominator;
uint256 rewardPoolFee = (fees * 200 ) / feeDenominator;
uint256 lpProviderFee = (fees * 100 ) / feeDenominator;
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L90,99
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
lPTokenReceiver
liquidityPair
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L185
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(limitsInEffect)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

RCU - Redundant Check Usage

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L253,264
Status	Unresolved

Description

The contract is programmed to enforce limits on transactions by including a requirement statement in its functions that checks if the limit value is both less than the `feeDenominator` (1000) and less than the `maxSellFee` or `maxBuyFee` (both set to 10). This implementation introduces a redundancy in the checks. Since the `maxSellFee` and `maxBuyFee` values (10) are significantly smaller than the `feeDenominator` (1000), the condition `limit <= feeDenominator` is always true whenever `limit < maxSellFee` or `limit < maxBuyFee` holds. As a result, the `feeDenominator` check does not serve its intended purpose of being a meaningful constraint, which makes the logic and intent of the code unclear and potentially confusing.

```
require(
    limit <= feeDenominator && limit < maxSellFee,
    "Limit cannot exceeds the max sell fee of 10%"
);
...
require(
    limit <= feeDenominator && limit < maxBuyFee,
    "Limit cannot exceeds the max sell fee of 10%"
);
```

Recommendation

It is recommended to remove the check `limit <= feeDenominator` from the requirement statements, as it never effectively applies its logic under the current settings. Simplifying this condition to only `limit < maxSellFee` and `limit < maxBuyFee` will make the code cleaner and the intent of the limit enforcement clearer. This adjustment not only enhances the readability and maintainability of the smart contract code but also optimizes execution by removing unnecessary computation steps.

RFM - Redundant Fee Mechanism

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L297
Status	Unresolved

Description

The contract is designed to calculate and accumulate marketing and liquidity fees separately based on transactions. It specifies the use of distinct fee variables, to compute fees for marketing and liquidity respectively. These fees are then added to their corresponding pools, `tokensForMarketing` and `tokensForLiquidity` . However, the contract further processes the accumulated fees to calculate additional fees based on a percentage of the total accumulated fees. This results in a redundancy, as the initial separation into marketing and liquidity fees does not influence the subsequent calculations or fee distributions, but rather, all are derived from the total fees computed initially. This not only adds unnecessary complexity but also leads to inefficiencies in fee management and use.

```
   uint256 newTokensForMarketing = (amount * sellMarketingFee)
/ feeDenominator;
    uint256 newTokensForLiquidity = (amount * sellLiquidityFee)
/ feeDenominator;
    fees = newTokensForMarketing + newTokensForLiquidity;

    if (fees > 0) {
        uint256 adminFee = (fees * 500 ) / feeDenominator;
        uint256 burnFee = (fees * 200 ) / feeDenominator;
        uint256 boostLpFee = (fees * 100 ) / feeDenominator;
        uint256 lpProviderFee = (fees * 200 ) / feeDenominator;
    tokensForMarketing += newTokensForMarketing;
    tokensForLiquidity += newTokensForLiquidity;

} else if (isAMM[from] && buyTotalFees > 0) {
    uint256 newTokensForMarketing = (amount * buyMarketingFee)
/ feeDenominator;
    uint256 newTokensForLiquidity = (amount * buyLiquidityFee)
/ feeDenominator;
    fees = newTokensForMarketing + newTokensForLiquidity;

    if (fees > 0) {
        uint256 adminFee = (fees * 600 ) / feeDenominator;
        uint256 burnFee = (fees * 100 ) / feeDenominator;
        uint256 rewardPoolFee = (fees * 200 ) / feeDenominator;
        uint256 lpProviderFee = (fees * 100 ) / feeDenominator;
    ...
    tokensForMarketing += newTokensForMarketing;
    tokensForLiquidity += newTokensForLiquidity;
```

Recommendation

It is recommended to remove the marketing and liquidity fees since the contract does not base any functionality on these fees. Instead, the contract could calculate a general fee variable and then split the fee as needed. Simplifying the fee structure in this way will reduce complexity and potential errors, streamline transactions, and improve contract efficiency. This approach would allow the contract to maintain flexibility in fee allocation while reducing the overhead associated with managing multiple fee variables that ultimately serve no distinct purpose.

RFU - Redundant Functionalities Usage

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L129,191
Status	Unresolved

Description

The contract is designed to contain functions that allow for the updating of specific contract variables. However, these variables are not utilized in any significant operational functions or logical processes within the contract's implementation. The presence of such functions not only contributes to increased contract complexity but also introduces potential security risks. These include unnecessary exposure to address modification functions and the maintenance of variable states that do not contribute to the contract's functionality or behavior. The inclusion of these redundant functions and variables may lead to potential misinterpretations of the contract's intended capabilities and unnecessarily complicate its architecture.

```
swapThreshold = (totalSupply * 1000) / 1000;

function setMarketingReceiver(address newReceiver) external
onlyOwner {
    require(
        marketingReceiver != newReceiver,
        "Address is already set for marketing token receiver
address"
    );
    isExcludedFromFee[newReceiver] = true;
    isExcludedFromWalletLimits[newReceiver] = true;
    marketingReceiver = newReceiver;
    emit MarketingReceiverSet(newReceiver);
}
```

Recommendation

It is recommended to thoroughly review and remove such functions that update unused variables and any associated unused variables from the contract. Eliminating these redundancies will streamline the contract by removing unnecessary code, reducing the attack surface related to unused functionalities, and minimizing potential confusion regarding the contract's actual functionalities and their purposes. Simplifying the contract in this manner enhances its security, maintainability, and efficiency, aligning it more closely with its operational objectives.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L129
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
swapThreshold = (totalSupply * 1000) / 1000;
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L58
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
bool private swapping
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L204,205,206,207
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _adminWallet
address _boostLiquidityWallet
address _lpProviderWallet
address _rewardsPoolWallet
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/Mforge.sol#L209,210,211,212
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
adminWallet = _adminWallet
boostLiquidityWallet = _boostLiquidityWallet
lpProviderWallet = _lpProviderWallet
rewardsPoolWallet = _rewardsPoolWallet
```

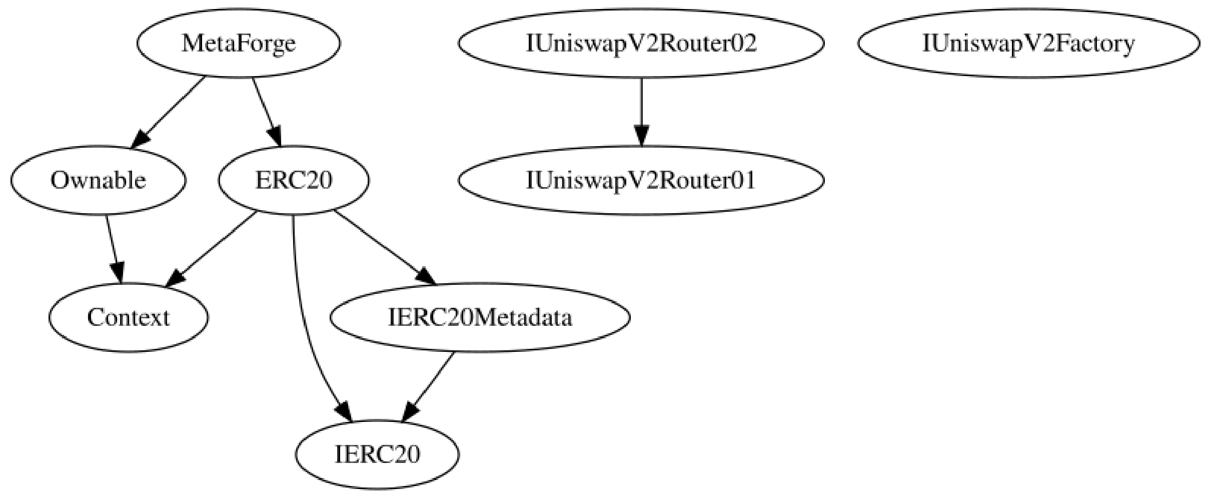
Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

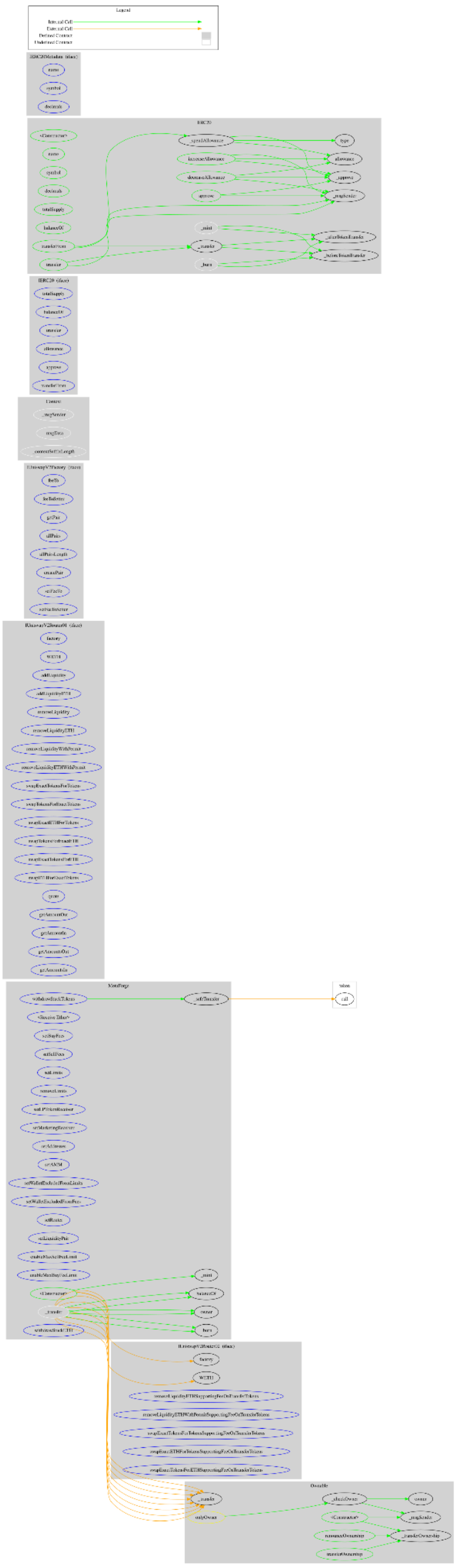
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
MetaForge	Implementation	ERC20, Ownable		
		Public	✓	ERC20
		External	Payable	-
	setBuyFees	External	✓	onlyOwner
	setSellFees	External	✓	onlyOwner
	removeLimits	External	✓	onlyOwner
	setMarketingReceiver	External	✓	onlyOwner
	setAddresses	External	✓	onlyOwner
	setAMM	External	✓	onlyOwner
	setWalletExcludedFromLimits	External	✓	onlyOwner
	setWalletExcludedFromFees	External	✓	onlyOwner
	enableMaxSellFeeLimit	External	✓	onlyOwner
	enableMaxBuyFeeLimit	External	✓	onlyOwner
	_transfer	Internal	✓	
	withdrawStuckTokens	External	✓	-
	withdrawStuckETH	External	✓	-
	_safeTransfer	Private	✓	

Inheritance Graph



Flow Graph



Summary

MagnumBit contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. There is also a limit of max 2% fees.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io