Cyberscope

Audit Report

MyUsdRewardPool

March 2023

Network BSC

Address 0x51a93E1484C3562632d4e480ffc9BF4ac1AFe64D

Audited by © cyberscope

Table of Contents

Review

Explorer	https://bscscan.com/address/0x51a93e1484c3562632d4e480ffc9bf4ac1afe64d

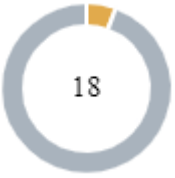
Audit Updates

Initial Audit	03 Apr 2023

Source Files

Filename	SHA256
MyUSDRewardPool.sol	ef7807993e26700bfbfeddd5b03694cce4281e5c76a7dfa54206b3d7b6228942

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	17

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	17	0	0	0

Introduction

MyUsdRewardPool implements a staking contract. It allows users to stake their liquidity pool tokens and earn rewards. The Contract operates on a yield farming mechanism, where users can earn additional rewards for providing liquidity for longer periods.

Roles

The contract roles consist of the operator role.

The `operator` has the authority to:

- Add a new lp to the pool.
- Update the given pool's allocation points.
- Change operator address.
- Update pool start time.
- Recover unsupported tokens.

The users have the authority to:

- View accumulate rewards over the given period.
- View pending shares.
- Update reward variables for all pools.
- Update reward variables of the given pool.
- Deposit LP tokens.
- Withdraw LP tokens.
- Withdraw LP tokens without rewards.

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	PSTI	Pool Start Time Inconsistency	Unresolved
Minor / Informative	PRI	PendingShare Reward Inconsistency	Unresolved
Minor / Informative	WLT	Withdraw Liquidity Tokens	Unresolved
Minor / Informative	URI	User Rewards Inconsistency	Unresolved
Minor / Informative	BPV	Block Property Vulnerability	Unresolved
Minor / Informative	AAO	Accumulated Amount Overflow	Unresolved
Minor / Informative	MCM	Misleading Comment Messages	Unresolved
Minor / Informative	RSK	Redundant Storage Keyword	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	L02	State Variables could be Declared Constant	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L06	Missing Events Access Control	Unresolved
Minor / Informative	L07	Missing Events Arithmetic	Unresolved

●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L18	Multiple Pragma Directives	Unresolved

PTAI - Potential Transfer Amount Inconsistency

Criticality	Medium
Location	MyUSDRewardPool.sol#L
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
function safeMyUSDTransfer(address _to, uint256 _amount)
internal {
    uint256 _myUSDBal = myUSD.balanceOf(address(this));
    if (_myUSDBal > 0) {
        if (_amount > _myUSDBal) {
            myUSD.safeTransfer(_to, _myUSDBal);
        } else {
            myUSD.safeTransfer(_to, _amount);
        }
    }
}
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

PSTI - Pool Start Time Inconsistency

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L726
Status	Unresolved

Description

The `updatePool` function initializes LQ pools without checking the `lastRewardTime` for unstarted pools, which can lead to inconsistencies in their start times. Moreover, it can cause incorrect reward calculations for pools that have already started.

```
function updatePool(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    if (block.timestamp <= pool.lastRewardTime) {
        return;
    }
    uint256 tokenSupply = pool.token.balanceOf(address(this));
    if (tokenSupply == 0) {
        pool.lastRewardTime = block.timestamp;
        return;
    }
    if (!pool.isStarted) {
        pool.isStarted = true;
        totalAllocPoint = totalAllocPoint.add(pool.allocPoint);
    }
    if (totalAllocPoint > 0) {
        uint256 _generatedReward =
getGeneratedReward(pool.lastRewardTime, block.timestamp);
        uint256 _myUSDReward =
_generatedReward.mul(pool.allocPoint).div(totalAllocPoint);
        pool.accMyUSDPerShare =
pool.accMyUSDPerShare.add(_myUSDReward.mul(1e18).div(tokenSuppl
y));
    }
    pool.lastRewardTime = block.timestamp;
}
```

Recommendation

It is recommended to ensure that the LP pool starts at the right time to ensure that the rewards are calculated correctly.

PRI - PendingShare Reward Inconsistency

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L704
Status	Unresolved

Description

The `pendingShare` function is responsible for calculating the pending rewards for a user. However, it fails to correctly aggregate the rewards, resulting in an incorrect reward calculation for the user. This inconsistency in reward calculation is caused by inconsistencies in the pool start time. Since incorrect total allocation might be added to the variable `totalAllocPoint`.

```
function pendingShare(uint256 _pid, address _user) external
view returns (uint256) {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][_user];
    uint256 accMyUSDPerShare = pool.accMyUSDPerShare;
    uint256 tokenSupply = pool.token.balanceOf(address(this));
    if (block.timestamp > pool.lastRewardTime && tokenSupply !=
0) {
        uint256 _generatedReward =
getGeneratedReward(pool.lastRewardTime, block.timestamp);
        uint256 _myUSDReward =
_generatedReward.mul(pool.allocPoint).div(totalAllocPoint);
        accMyUSDPerShare =
accMyUSDPerShare.add(_myUSDReward.mul(1e18).div(tokenSupply));
    }
    return
user.amount.mul(accMyUSDPerShare).div(1e18).sub(user.rewardDebt
);
}
```

Recommendation

To address this issue, it is recommended to resolve the inconsistencies in the pool start time. Doing so will ensure that the `pendingShare` function correctly calculates the rewards for users, improving the overall accuracy of the reward system.

WLT - Withdraw Liquidity Tokens

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L823
Status	Unresolved

Description

The contract allows the owner to withdraw liquidity tokens until 90 days after the pool has ended, which may create a security vulnerability if the owner decides to withdraw liquidity without user consent or in a way that negatively impacts the users.

```
function governanceRecoverUnsupported(IERC20 _token, uint256
amount, address to) external onlyOperator {
    if (block.timestamp < poolEndTime + 90 days) {
        // do not allow to drain core token (MyUSD or lps) if
less than 90 days after pool ends
        require(_token != myUSD, "myUSD");
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            require(_token != pool.token, "pool.token");
        }
    }
    _token.safeTransfer(to, amount);
}
```

Recommendation

It is recommended to remove the ability of the operator to withdraw liquidity tokens from the pool. This can help ensure the stability of the pool and prevent any potential manipulation or abuse.

URI - User Rewards Inconsistency

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L801
Status	Unresolved

Description

If a contract does not have a sufficient balance of tokens, then it may not be able to distribute the expected rewards to users. This can lead to users receiving less than the expected amount and may result in dissatisfaction or loss of trust in the contract.

```
function safeMyUSDTransfer(address _to, uint256 _amount)
internal {
    uint256 _myUSDBal = myUSD.balanceOf(address(this));
    if (_myUSDBal > 0) {
        if (_amount > _myUSDBal) {
            myUSD.safeTransfer(_to, _myUSDBal);
        } else {
            myUSD.safeTransfer(_to, _amount);
        }
    }
}
```

Recommendation

It is recommended to regularly monitor the contract balance and ensure that it is sufficient to distribute the expected rewards to users. Additionally, implementing a failsafe mechanism to prevent distribution if the balance is too low can also help mitigate this issue.

BPV - Block Property Vulnerability

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol
Status	Unresolved

Description

The contract is currently using the block.timestamp variable to determine the pool start time. However, this value can be manipulated by miners to a certain extent, which can potentially create security vulnerabilities in the contract.

```
if (block.timestamp < poolStartTime) {
if (_lastRewardTime == 0 || _lastRewardTime < block.timestamp)
{
    _lastRewardTime = block.timestamp;
}
(_lastRewardTime <= block.timestamp);
...
```

Recommendation

It is recommended to use the block.number function instead of block.timestamp, as it is much more difficult to manipulate the block number. By using block.number, the contract can still maintain the time-based functionality without exposing itself to manipulation risks.

AAO - Accumulated Amount Overflow

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L592
Status	Unresolved

Description

The contract is using variables to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

The variable `totalAllocPoint` accumulates values.

```
uint256 public totalAllocPoint = 0;
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum value that can be stored in its data type.

MCM - Misleading Comment Messages

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L600,601
Status	Unresolved

Description

The contract is using misleading comment messages. These comment messages do not accurately reflect the actual implementation, making it difficult to understand the source code. As a result, the users will not comprehend the source code's actual implementation.

```
uint256 public MyUSDPerSecond = 3805175038051750; // 120,000
myUSD per year:: 120,000/ (365 days * 24h * 60min * 60s)
uint256 public runningTime = 1095 days; // 365 days
```

Recommendation

The team is advised to carefully review the comment in order to address these issues. To improve code readabilty, the team should use more specific and descriptive comment messages.

RSK - Redundant Storage Keyword

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L701,702
Status	Unresolved

Description

The contract uses the `storage` keyword in a view function. The `storage` keyword is used to persist data on the contract's storage. View functions are functions that do not modify the state of the contract and do not perform any actions that cost gas (such as sending a transaction). As a result, the use of the `storage` keyword in view functions is redundant.

```
PoolInfo storage pool
UserInfo storage user
```

Recommendation

It is generally considered good practice to avoid using the `storage` keyword in view functions, because it is unnecessary and can make the code less readable.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L614
Status	Unresolved

Description

The contract is using variables that initialize them only in the constructor. The other functions are not mutating the variables. These variables are not defined as `immutable`.

```
myUSD
```

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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L600,601
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.

```
uint256 public MyUSDPerSecond = 1902587519025875
uint256 public runningTime = 1095 days
```

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	MyUSDRewardPool.sol#L600,625,634,635,636,637,674,686,700,722,745,766,786,797,808,813,819
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.

```
uint256 public MyUSDPerSecond = 1902587519025875
IERC20 _token
uint256 _allocPoint
bool _withUpdate
uint256 _lastRewardTime
uint256 _pid
uint256 _toTime
uint256 _fromTime
address _user
uint256 _amount
address _to
address _operator
uint256 _newPoolStartTime
```

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.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L810
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
operator = _operator
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L669,678,815
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
totalAllocPoint = totalAllocPoint.add(_allocPoint)

totalAllocPoint = totalAllocPoint.sub(pool.allocPoint).add(
                _allocPoint
            )
poolStartTime = _newPoolStartTime
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L133,159,184,209,219,233,243,515,526,531
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function sendValue(address payable recipient, uint256 amount)
internal {
        require(address(this).balance >= amount, "Address:
insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls,
avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value,
recipient may have reverted");
...
function functionCall(address target, bytes memory data)
internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level
call failed");
    }

function functionCallWithValue(address target, bytes memory
data, uint256 value) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value,
"Address: low-level call with value failed");
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L707,708,738,739
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 _myUSDReward =
_generatedReward.mul(pool.allocPoint).div(totalAllocPoint)
accMyUSDPerShare =
accMyUSDPerShare.add(_myUSDReward.mul(1e18).div(tokenSupply))
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L113,260
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly { size := extcodesize(account) }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	MyUSDRewardPool.sol#L6,83,556
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity >=0.6.0 <0.8.0;
pragma solidity >=0.6.2 <0.8.0;
pragma solidity 0.6.12;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

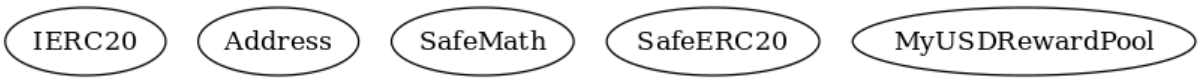
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	

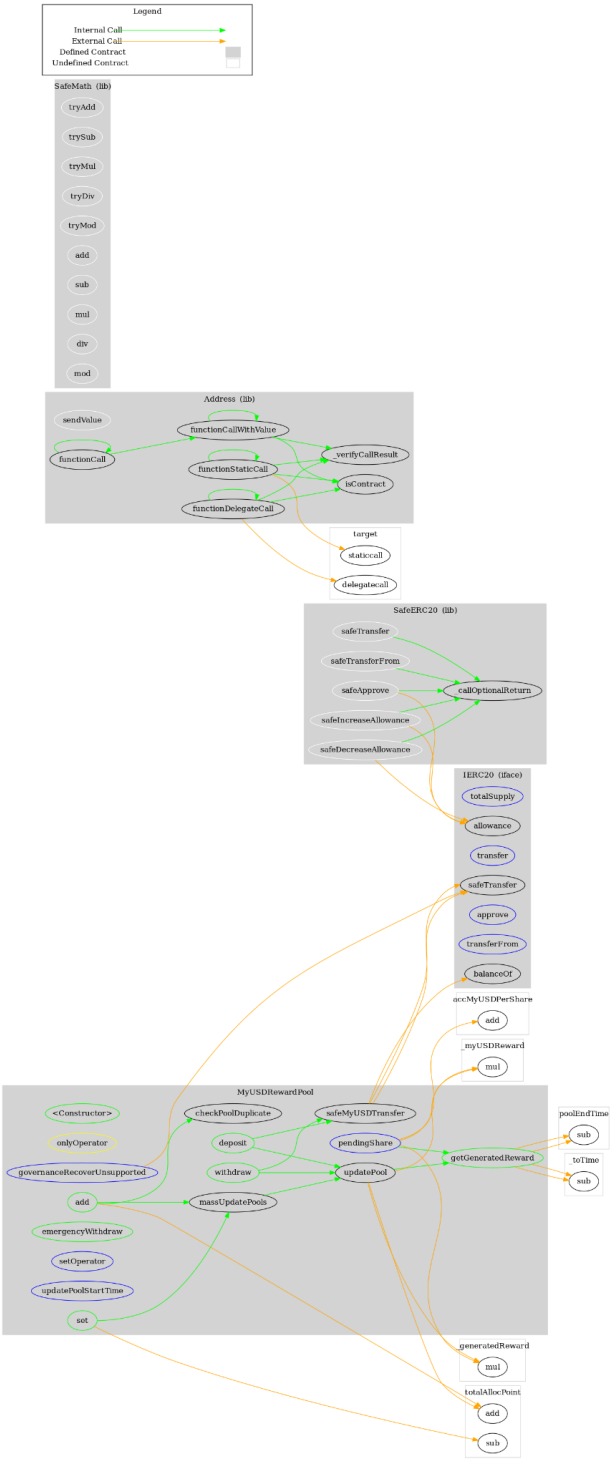
	functionDelegateCall	Internal	✓	
	_verifyCallResult	Private		
SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		
SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	safeIncreaseAllowance	Internal	✓	

	safeDecreaseAllowance	Internal	✓	
	_callOptionalReturn	Private	✓	
MyUSDReward Pool	Implementation			
		Public	✓	-
	checkPoolDuplicate	Internal		
	add	Public	✓	onlyOperator
	set	Public	✓	onlyOperator
	getGeneratedReward	Public		-
	pendingShare	External		-
	massUpdatePools	Public	✓	-
	updatePool	Public	✓	-
	deposit	Public	✓	-
	withdraw	Public	✓	-
	emergencyWithdraw	Public	✓	-
	safeMyUSDTransfer	Internal	✓	
	setOperator	External	✓	onlyOperator
	updatePoolStartTime	External	✓	onlyOperator
	governanceRecoverUnsupported	External	✓	onlyOperator

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
SafeMath (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
add
sub
mul
div
mod
Address (lib)
sendValue
functionCallWithValue
functionCall
_verifyCallResult
functionStaticCall
isContract
functionDelegateCall
target
staticcall
delegatecall
SafeERC20 (lib)
safeTransfer
safeTransferFrom
safeApprove
callOptionalReturn
safeIncreaseAllowance
safeDecreaseAllowance
IERC20 (iface)
totalSupply
allowance
transfer
safeTransfer
approve
transferFrom
balanceOf
accMyUSDPerShare
add
myUSDReward
mul
MyUSDRewardPool
<Constructor>
checkPoolDuplicate
safeMyUSDTransfer
onlyOperator
deposit
pendingShare
governanceRecoverUnsupported
withdraw
updatePool
getGeneratedReward
add
massUpdatePools
emergencyWithdraw
setOperator
updatePoolStartTime
set
poolEndTime
sub
toTime
sub
generatedReward
mul
totalAllocPoint
add
sub

Summary

MUsdRewardPool contract implements a staking mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

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