Cyberscope

Audit Report

Pocketcoin Staking

March 2025

SHA256

23bd0e996a3c04ce91f5025115ea5555eb14652b588c4e8197434d41dac2f40f

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

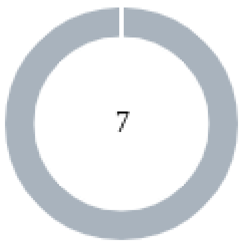
Audit Updates

Initial Audit	27 Feb 2025 https://github.com/cyberscope-io/audits/blob/main/pkoin/v1/audit.pdf
Corrected Phase 2	12 Mar 2025
Test Deploy	https://sepolia.etherscan.io/address/0x00D2F8c99a62093bE1F9Af12E85Ad750e14A61D0

Source Files

Filename	SHA256
StakingContract.sol	23bd0e996a3c04ce91f5025115ea5555eb14652b588c4e8197434d41dac2f40f

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	7

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	3	4	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CCS	Commented Code Segments	Unresolved
●	CCR	Contract Centralization Risk	Acknowledged
●	CRP	Coupled Rewards Pool	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
●	RCS	Redundant Conditional Statements	Unresolved
●	UM	Undefined Multiplier	Unresolved
●	L19	Stable Compiler Version	Acknowledged

CCS - Commented Code Segments

Criticality	Minor / Informative
Location	StakingContract.sol#L323
Status	Unresolved

Description

The contract contains several code segments that are commented out. Blocks of code, including important operations and validation checks, are present but commented out. Commented code can be a source of confusion, as it's unclear whether these segments are meant for future use, are remnants of previous iterations, or are temporarily disabled for testing purposes. Moreover, commented out code can clutter the contract, making it more challenging to read and understand the actual functioning code.

```
// OR Introduce a penalty for early withdrawal of funds.
// Update the totalPlannedRewards value only if the user was
eligible for the full reward.
/*
uint256 penalty = (userStake.amount * 5) / 100; // 5% penalty
uint256 returnAmount = userStake.amount - penalty;
token.safeTransfer(msg.sender, returnAmount);
token.safeTransfer(owner, penalty);
*/
```

Recommendation

It is recommended to either remove the parts of the code that are not intended to be used or to declare and code the appropriate segments properly if they are meant for future implementation. If the intention is to preserve these segments for historical or reference purposes, it would be beneficial to move them to documentation outside of the active codebase. This approach helps maintain the clarity and cleanliness of the contract's code, ensuring that it accurately reflects its current functionality and intended use.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	StakingContract.sol#L333,353,375,383
Status	Acknowledged

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function withdrawRemainingRewards() external nonReentrant onlyOwner
{...}
function proposeMultiplierUpdate(uint256 lockDuration, uint256
newMultiplier) external onlyOwner {...}
function executeMultiplierUpdate(uint256 lockDuration) external
onlyOwner {...}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

CRP - Coupled Rewards Pool

Criticality	Minor / Informative
Location	StakingContract.sol#L225
Status	Acknowledged

Description

The contract maintains a balance of staked tokens and tokens aimed for rewards, where the staked tokens and reward tokens are the same token. The contract accepts a new stake by comparing its current balance to the staked amounts and the incoming deposit and rewards. The deposited balance of previous stakes is used to determine the eligibility of future stakes. Coupling user deposits with the expected rewards and the contract’s balance is prone to errors and could yield inconsistencies.

```
uint256 contractBalance = token.balanceOf(address(this));
require(
contractBalance >= (totalStaked + totalPlannedRewards + amount +
plannedReward),
"Insufficient rewards in contract"
);
```

Recommendation

It is advisable to ensure consistency by maintaining a strict association between the available reward pool and the expected rewards. Decoupling the staked balance from the reward pool will ensure overall consistency.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	StakingContract.sol#L239
Status	Acknowledged

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

```
// Record the stake
stakes[msg.sender].push(
    Stake({
    amount: amount,
    startTime: block.timestamp,
    lockDuration: lockDuration,
    multiplier: multiplier,
    withdrawn: false,
    emergencyWithdrawn: false
    })
);

// Transfer tokens from user to contract
token.safeTransferFrom(msg.sender, address(this), 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
```

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	StakingContract.sol#L117
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Conditional statements that are always satisfied lead to larger code size, increased memory usage, and slower execution times. By directly removing redundancies the code can be made more concise and efficient, reducing gas costs and improving runtime performance.

```
require(msg.sender != address(0), "Owner cannot be zero address");
```

Recommendation

It is recommended to refactor conditional statements that are always satisfied by eliminating unnecessary code structures. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage.

UM - Undefined Multiplier

Criticality	Minor / Informative
Location	StakingContract.sol#L375
Status	Unresolved

Description

The contract implements the `addAllowedDuration` method, enabling the owner to extend the current functionality with new locking durations. However, a multiplier is not defined at the time of the new duration's definition. This omission may lead to confusion for users, as the default multiplier will be set to zero.

```
function addAllowedDuration(uint256 newDuration) external onlyOwner
{
require(!isAllowedDuration[newDuration], "Duration already
allowed");
isAllowedDuration[newDuration] = true;
allowedDurations.push(newDuration);
}
```

Recommendation

It is advised to ensure consistency by simultaneously introducing a locked duration and the respective multiplier, while also performing proper checks on the introduced values. This approach helps prevent user confusion and ensures that each new duration has an appropriate multiplier defined from the onset.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	StakingContract.sol#L58
Status	Acknowledged

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.26;
```

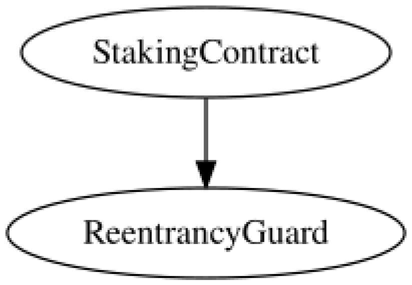
Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

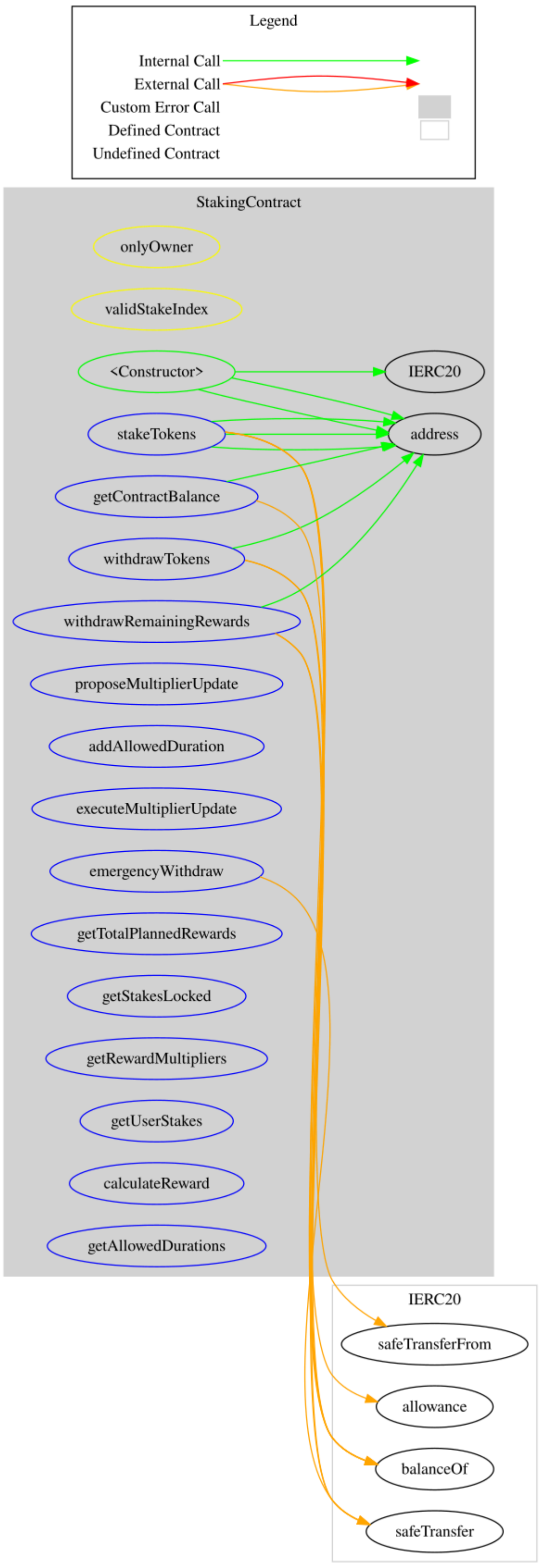
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
StakingContract	Implementation	ReentrancyGuard		
		Public	✓	-
	stakeTokens	External	✓	nonReentrant
	withdrawTokens	External	✓	nonReentrant validStakeIndex
	emergencyWithdraw	External	✓	nonReentrant validStakeIndex
	withdrawRemainingRewards	External	✓	nonReentrant onlyOwner
	proposeMultiplierUpdate	External	✓	onlyOwner
	addAllowedDuration	External	✓	onlyOwner
	executeMultiplierUpdate	External	✓	onlyOwner
	getContractBalance	External		-
	getTotalPlannedRewards	External		-
	getStakesLocked	External		-
	getRewardMultipliers	External		-
	getUserStakes	External		-
	calculateReward	External		validStakeIndex
	getAllowedDurations	External		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
StakingContract
onlyOwner
validStakeIndex
<Constructor>
IERC20
stakeTokens
address
getContractBalance
withdrawTokens
withdrawRemainingRewards
proposeMultiplierUpdate
addAllowedDuration
executeMultiplierUpdate
emergencyWithdraw
getTotalPlannedRewards
getStakesLocked
getRewardMultipliers
getUserStakes
calculateReward
getAllowedDurations
IERC20
safeTransferFrom
allowance
balanceOf
safeTransfer

Summary

Pocketcoin contract implements a staking mechanism. This audit investigates security issues, business logic concerns and potential improvements. The Smart Contract analysis reported no compiler error or critical issues.

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

cyberscope.io