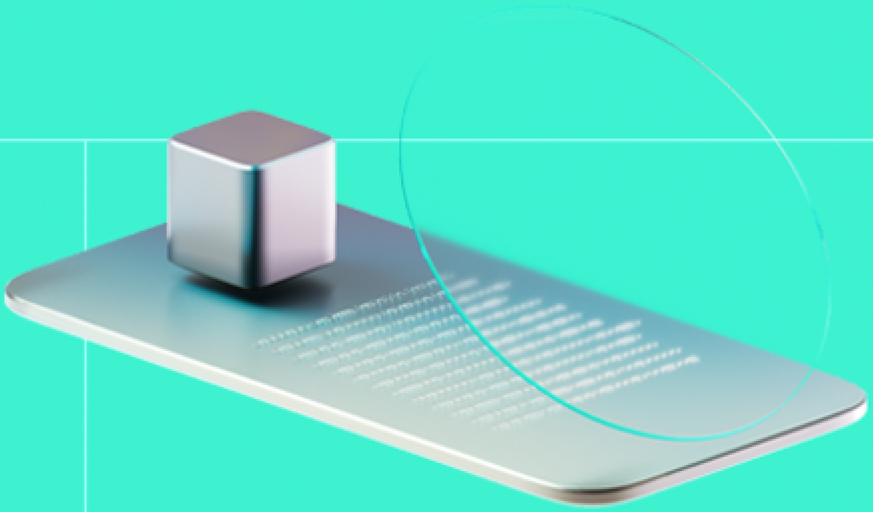
Smart Contract Code Review And Security Analysis Report



Customer: Pikamoon

Date: 14/05/2024

We express our gratitude to the Pikamoon team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Pikamoon is a staking smart contract.

Platform: EVM

Language: Solidity

Tags: Staking

Timeline: 03/05/2024 - 14/05/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/orbit-cosmos/pikamoon-staking
Commit	86cf91c3e9d5f89fa4a16bea22b5cae262019a1a

Audit Summary

10/10	10/10	80.68%	10/10
Security score	Code quality score	Test coverage	Documentation quality score

Total 9.3/10

The system users should acknowledge all the risks summed up in the risks section of the report

0	0	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	0
Low	0

Document

Name	Smart Contract Code Review and Security Analysis Report for Pikamoon
Audited By	Max Fedorenko, Roman Tiutiun
Approved By	Grzegorz Trawiński, Ataberk Yavuzer
Website	https://www.pikamoon.io/
Changelog	08/05/2024 - Preliminary Report
	13/05/2024 - Second Review Report
	14/05/2024 - Final Report

Table of Contents

System Overview

Pikamoon is a staking protocol with the following contracts:

- CorePool — a contract that manages token staking.
- PikaStakingPool — is an upgradeable implementation of a staking pool for a token.
- PoolController — manages staking pools and holds the rewards.
- Stake — library responsible for the weight calculation and storing important constants related to stake period, base weight, and multipliers utilized.

Privileged roles

- The owner of the `CorePool.sol` contract can pause/unpause the staking contract, and set the address which is responsible for approving the users staking rewards.
- The owner of the `PoolController.sol` contract can register a new pool, update the rate of PIKA distribution per second, changes the weight of the pool.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation quality score is **10** out of **10**.

- NatSpec covers the code and is sufficient.

Code quality

The total Code quality score is **10** out of **10**.

Test coverage

Code coverage of the project is **80.68%** (branch coverage).

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **0** medium, and **0** low severity issues, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.3**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- **Single Points of Failure and Control**: The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- **Absence of Time-lock Mechanisms for Critical Operations**: Without time-locks on critical operations, there is no buffer to review or revert potentially harmful actions, increasing the risk of rapid exploitation and irreversible changes.
- **Administrative Key Control Risks**: The digital contract architecture relies on administrative keys for critical operations. Centralized control over these keys presents a significant security risk, as compromise or misuse can lead to unauthorized actions or loss of funds.
- **Single Entity Upgrade Authority**: The contract grants a single entity the authority to implement upgrades or changes. This centralization of power risks unilateral decisions that may not align with the community or stakeholders' interests, undermining trust and security.
- **Absence of Upgrade Window Constraints**: The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.
- **Flexibility and Risk in Contract Upgrades**: The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.

Findings

Vulnerability Details

Observation Details

F-2024-2017 - Code Contains `hardhat` Commented Import - Info

Description: In Solidity development, `hardhat` is a commonly used development framework that provides tools for testing, debugging, and deploying smart contracts. However, imports from `hardhat` in your Solidity code, such as those used for testing or local development, should not make their way into the final production version of the contract. These imports can bloat the contract, lead to unnecessary complexity, and potentially introduce security risks or unexpected behavior. Ensuring that production code is clean, efficient, and free of development-only dependencies is crucial for maintaining security and performance.

```
// import "hardhat/console.sol";
```

Assets:

- CorePool.sol [https://github.com/orbit-cosmos/pikamoon-staking]

Status: Fixed

Recommendations

Remediation: Before finalizing and deploying your Solidity contracts, thoroughly review the code to ensure all `hardhat` imports and related development-only code are removed.

Resolution: The `hardhat` import was removed from the code in commit **d677254**.

F-2024-2356 - Redundant `_msgSender()`, Meta-Transactions Not Implemented - Info

Description: The `_msgSender()` function is needed to handle the meta transactions, in the OpenZeppelin library, it is used to support the development of the libraries which might be used with the contracts with the specified TrustedForwarder or to be used in such contracts directly.

However, the current implementation is not a library and does not rewrite the `_msgSender()` function to support meta transactions. This leads to the redundancy, because the system do not utilize `_msgSender()` features and used only as a substitute for `msg.sender`.

Assets:

- CorePool.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- PoolController.sol [https://github.com/orbit-cosmos/pikamoon-staking]

Status: Fixed

Recommendations

Remediation: If `_msgSender()` is not going to be used in current implementation to support the meta transactions it is recommended to use `msg.sender` to prevent unforeseen issues and misinterpretations that may occur after the future updates.

Resolution: The `_msgSender()` was replaced to `msg.sender` in commit **d677254**.

F-2024-2368 - Missing Events Emitting For Critical Functions - Info

Description:	The following function does not emit relevant event after executing the sensitive action of setting the `verifierAddress`. This makes off-chain tracking of crucial state changes more complex.
Assets:	• PoolController.sol [https://github.com/orbit-cosmos/pikamoon-staking]
Status:	Fixed

Recommendations

Remediation:	Consider emitting event after sensitive change occur to simplify off-chain tracking of the contract state.
Resolution:	The event emitting has been added to the aforementioned asset in commit **d677254**.

F-2024-2708 - Solidity version 0.8.20 might not work on all chains due to `PUSH0` - Info

Description: The Solidity version 0.8.20 employs the recently introduced `PUSH0` opcode in the Shanghai EVM. This opcode might not be universally supported across all blockchain networks and Layer 2 solutions. Thus, as a result, it might be not possible to deploy a solution with version 0.8.20 >= on some blockchains.

```
pragma solidity 0.8.20;
```

Assets:

- CorePool.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- PikaStakingPool.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- PoolController.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- libraries/Errors.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- libraries/Stake.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- interfaces/ICorePool.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- interfaces/IPikaMoon.sol [https://github.com/orbit-cosmos/pikamoon-staking]
- interfaces/IPoolController.sol [https://github.com/orbit-cosmos/pikamoon-staking]

Status: Accepted

Recommendations

Remediation: It is recommended to verify whether a solution can be deployed on a particular blockchain with the Solidity version 0.8.20 >=. Whenever such deployment is not possible due to a lack of `PUSH0` opcode support and lowering the Solidity version is a must, it is strongly advised to review all feature changes and bugfixes in [Solidity releases] (https://soliditylang.org/blog/category/releases/). Some changes may have an impact on the current implementation and may impose a necessity of maintaining another version of the solution.

Resolution: The client acknowledges the risks and is responsible for ensuring compatibility with the specific blockchain on which the code is deployed.

F-2024-2709 - Best practice violation due to usage of assert() - Info

Description: The usage of `assert()` in `stake()`, `claimRewards()` functions are a violation of Solidity best practices, lowering the contract's code quality. Quoting from Solidity Language Description document:

```
The assert function creates an error of type Panic(uint256).
The same error is created by the compiler in certain situations as l
isted below.

Assert should only be used to test for internal errors, and to check
invariants.
Properly functioning code should never create a Panic, not even on i
nvalid external input.
If this happens, then there is a bug in your contract which you shou
ld fix.
```

Assets:

- CorePool.sol [https://github.com/orbit-cosmos/pikamoon-staking]

Status: Fixed

Recommendations

Remediation: It is recommended to replace `assert` check with `require` statement with meaningful error message or `if` statement with Custom Error.

Resolution: The `assert` statements were replaced with the `require` statement within the commit **61a7020**.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/orbit-cosmos/pikamoon-staking
Commit	86cf91c3e9d5f89fa4a16bea22b5cae262019a1a
Whitepaper	N/A
Requirements	https://github.com/orbit-cosmos/pikamoon-staking/tree/main/docs
Technical Requirements	https://github.com/orbit-cosmos/pikamoon-staking/tree/main/docs

Contracts in Scope

./contracts/CorePool.sol

./contracts/PikaStakingPool.sol

./contracts/PoolController.sol

./contracts/libraries/Errors.sol

./contracts/libraries/Stake.sol

./contracts/interfaces/ICorePool.sol

./contracts/interfaces/IPikaMoon.sol

./contracts/interfaces/IPoolController.sol