SMART CONTRACTS REVIEW

zokyo

June 24th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING PROJECT PI

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 3 High issues: 3 resolved = 0 points deducted
- 0 Medium issues: 0 points deducted
- 3 Low issues: 3 resolved = 0 points deducted
- 6 Informational issues: 6 resolved = 0 points deducted

Thus, the score is 100.

TECHNICAL SUMMARY

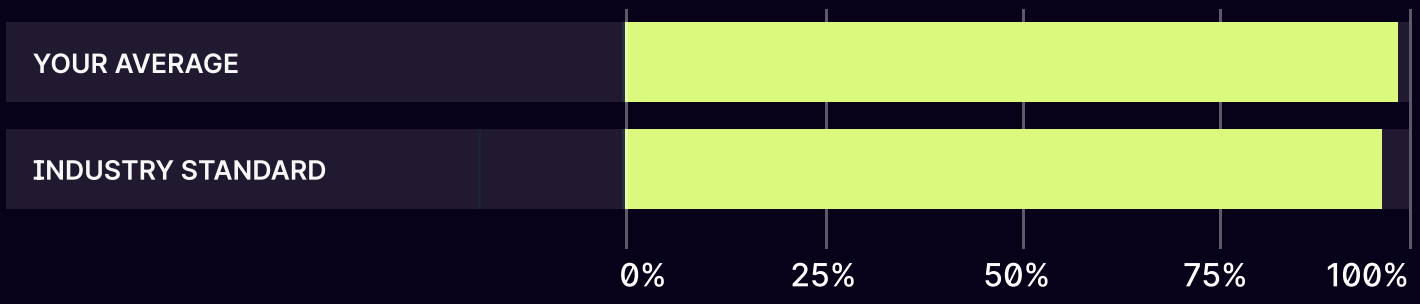
This document outlines the overall security of the Project Pi smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Project Pi smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

Testable Code



96.57% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Project Pi team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Project Pi repository:
Repo: https://github.com/VandaleCallender/ProjectPiSmartContracts/blob/main/MinipoolManager.sol

Last commit: 97c29837a1d881f2690befa59504d95966877c17

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- MinipoolManager.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Project Pi smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Testing contract/s logic against common and uncommon attack vectors.
04	Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

Pi Staking is the first permissionless staking protocol built for PulseChain, enabling node operators to launch more cheaply and quickly using the PPY token.

The Pi codebase for the audit consists of a contract called MinipoolManager which allows node operators to create and manage the Minipools.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Project Pi team and the Project Pi team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Incorrect Check For Early Stakers	High	Resolved
2	Incorrect Check To Verify startTime	High	Resolved
3	Incorrect check in the createMinipool() function	High	Resolved
4	Lack of Checks-effects-interactions pattern	Low	Resolved
5	Missing Zero Address Checks in Constructor	Low	Resolved
6	Missing Event Emissions	Low	Resolved
7	Unnecessary Check	Informational	Resolved
8	Public Functions Could Be Marked External	Informational	Resolved
9	Unused Variables	Informational	Resolved
10	Unnecessary converting	Informational	Resolved
11	Some variables could be made immutable	Informational	Resolved
12	Unnecessary if condition	Informational	Resolved

Incorrect Check For Early Stakers

For the first 99 stakers the minimum staking amount is 8e6 ether tokens (compared to 16e6 ether tokens for non-early stakers) , the code which verifies if the staker is an early staker is →

```
function canUseReducedStakingAmount(address stakerAddress) public view
returns (bool eligible, uint256 requiredStakingAmount) {
    EarlyStaking earlyStaking =
EarlyStaking(payable(getContractAddress("EarlyStaking")));
    int256 stakerIndex = earlyStaking.getIndexOf(stakerAddress);
    bool isEarlyStaker = stakerIndex >= 0; // True if stakerIndex is not
-1, meaning the staker exists

    // Assuming the first early staker gets a special treatment and should
use the reduced amount
    if (isEarlyStaker) {
        if (stakerIndex == 99) {
            // First 100 early stakers, eligible for reduced amount
            return (true, 8_000_000 ether);
        }
```

It checks if the stakerIndex == 99 , which is incorrect , this line would only be applicable for the 100th staker while it should be applicable to the first 100 stakers.

Recommendation:

Change the check to:

If (stakerIndex <= 99) return (true , 8_000_000 ether);

HIGH-2 | RESOLVED

Incorrect Check To Verify startTime

The start time for staking is assigned in the function recordStakingStart() , to verify if the startTime is a valid timestamp the following check is performed →

```
if (startTime > block.timestamp) {
        revert("InvalidStartTime");
    }
```

But this would be incorrect in a general sense since startTime can be higher than the current timestamp but it should not be less than the current timestamp which would be problematic . If the staking begins in the next block it should be fine since the end timestamp would be

Recommendation:

Change the check to

```
if (startTime < block.timestamp) {
        revert("InvalidStartTime");
    }
```

HIGH-3 | RESOLVED

Incorrect check in the createMinipool() function

MinipoolManager.sol L244
For the early stakers in the createMinipool() function, the function only accepts the combined contribution equal to the DAO's minimum requirement and plsAssignmentRequest equal to the DAO's max PLS assignment.

Recommendation:

Change the if condition to like this.

if (msg.value + plsAssignmentRequest < dao.getMinipoolMinPLSStakingAmt() || plsAssignmentRequest > dao.getMinipoolMaxPLSAssignment())

LOW-1 | RESOLVED

Lack of Checks-effects-interactions pattern

The Checks-Effects-Interactions (CEI) pattern is not followed in some parts of the smart contract. The CEI pattern is a well-known best practice in smart contract development that helps to prevent reentrancy attacks.

While the `nonReentrant` modifier is used in the contract, which effectively prevents reentrancy, it is still highly recommended to adhere to the CEI pattern in functions like `withdrawPartialRewards()`. Following the CEI pattern can prevent potential issues that might arise from future changes.

Recommendation:
Refactor the contract to implement the Checks-Effects-Interactions pattern.

LOW-2 | RESOLVED

Missing Zero Address Checks in Constructor

The constructor of the contract does not perform checks to ensure that the provided addresses are not zero addresses. It is important to validate that the addresses passed to the constructor are valid and not zero.

Recommendation:
Add checks in the constructor to ensure that the provided addresses are not zero addresses. This can be achieved using the require statement.

LOW-3 | RESOLVED

Missing Event Emissions

The functions `depositFromRewards()` and `receiveWithdrawalPLS()` do not emit events upon successful execution. Emitting events in smart contracts is crucial for tracking state changes and ensuring transparency. Events provide a reliable way to notify off-chain applications of changes and facilitate easier debugging and auditing.

Recommendation:
Consider emitting events in mentioned functions.

INFORMATIONAL-1 | RESOLVED

Unnecessary Check

The check “require(msg.sender == owner, "Only owner can withdraw");” on L357 is not needed since the onlyOwner() check at L356 is sufficient and would revert if msg.sender is not the owner.

Recommendation:
L357 can be removed

INFORMATIONAL-2 | RESOLVED

Public Functions Could Be Marked External

The `distributeRewards()` and `depositFromRewards` functions of the `MiniPoolManager.sol` contract is marked as public but it could be marked as external instead. While both public and external functions can be called from outside the contract, marking a function as external is generally more gas-efficient when it is not intended to be called internally.

Recommendation:
Consider changing the visibility of the claim function from public to external to optimize gas usage.

INFORMATIONAL-3 | RESOLVED

Unused Variables

The smart contract contains variables minStakingDuration and locked that are declared but not used anywhere in the code. Unused variables can lead to unnecessary gas costs and code bloat. They may also indicate incomplete functionality or leftover code from previous versions, which can be confusing and potentially error-prone.

Recommendation:
Remove the unused variables from the contract to optimize gas usage and improve code clarity.

INFORMATIONAL-4 | RESOLVED

Unnecessary converting

In the depositFromRewards() function, lastRewardsCycleEnd local variable is set with a value converted rewardsCycleEnd to uint32. But rewardsCycleEnd is already a uint32 type so no need to convert.

Recommendation:
Assign the lastRewardsCycleEnd with rewardsCycleEnd directly without converting.

INFORMATIONAL-5 | RESOLVED

Some variables could be made immutable

The nodeIDGenerator and validatorRegistration variables are only set in the constructor.

Recommendation:
Make the variables immutable.

Unnecessary if condition

In the createMinipool() function, if eligibleForReducedStaking is true, it checks if msg.value is greater than 8_000_000 ether. (L240)
But if eligibleForReducedStaking is true, then requiredStakingAmount will be 8_000_000 and msg.value is already checked with the requiredStakingAmount before.

Recommendation:
Remove the if condition.

	MinipoolManager.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Project Pi in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the Project Pi contract/s requirements for details about issuance amounts and how the system handles these.

Ran 49 tests for test/MinipoolManagerTest.t.sol:MinipoolManagerTest
[PASS] testTotalPLSLiquidStakerAmount() (gas: 28116)
[PASS] testNegativeCycleDuration() (gas: 369065)
[PASS] testWithdrawAmountTooLarge() (gas: 237291)
[PASS] testWithdrawForDelegationDisabled() (gas: 17992)
[PASS] testMinipoolCreation() (gas: 236429)
[PASS] testMinipoolCancellation() (gas: 308943)
[PASS] testMinipoolFundsWithdrawal() (gas: 302816)
[PASS] testMinipoolStatusChangeEvent() (gas: 575929)
[PASS] testWithdrawForDelegation() (gas: 439959)
[PASS] testDepositFromDelegation() (gas: 66926)
[PASS] testStakingClaimInitiationEligibility() (gas: 29975)
[PASS] testStakingClaimAndInitiation() (gas: 678901)
[PASS] testStakingStartRecording() (gas: 24670)
[PASS] testInvalidStakingStartTime() (gas: 236235)
[PASS] testStakingEndRecording() (gas: 589108)
[PASS] testStakingEndWithSlash() (gas: 123456)
[PASS] testStakingEndWithExcessiveSlashing() (gas: 23576)
[PASS] testStakingErrorRecording() (gas: 231476)
[PASS] testMultisigMinipoolCancellation() (gas: 29860)
[PASS] testExpectedRewardCalculation() (gas: 337216)
[PASS] testRetrieveMinipool() (gas: 35937)
[PASS] testRetrieveAllMinipools() (gas: 17912)
[PASS] testMinipoolCountRetrieval() (gas: 197211)
[PASS] testPPYSlashAmountCalculation() (gas: 236998)
[PASS] testFullCycleWithUserFunds() (gas: 14152)
[PASS] testFullCycleWithError() (gas: 222643)
[PASS] testInvalidMinipoolStateCycle() (gas: 21118)
[PASS] testMinipoolDurationExceededCycle() (gas: 262214)

We are grateful for the opportunity to work with the Project Pi team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Project Pi team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo