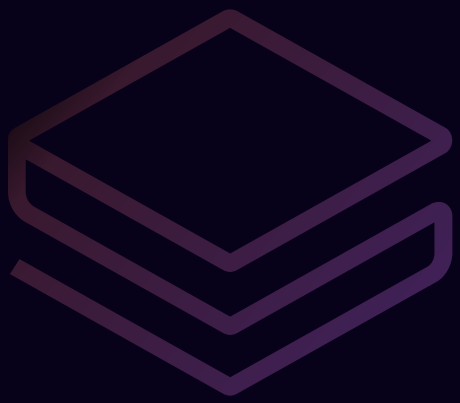


SMART CONTRACTS REVIEW

zokyo

March 25th 2024 | v. 1.0

Security Audit Score

PASS

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

ZOKYO AUDIT SCORING STRATIS GROUP LTD

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
- 0 High issues: 0 points deducted
- 0 Medium issues: 0 points deducted
- 1 Low issue: 1 resolved = 0 points deducted
- 5 Informational issues: 5 unresolved = - 5 points deducted

Thus, 100 - 5 = 95

TECHNICAL SUMMARY

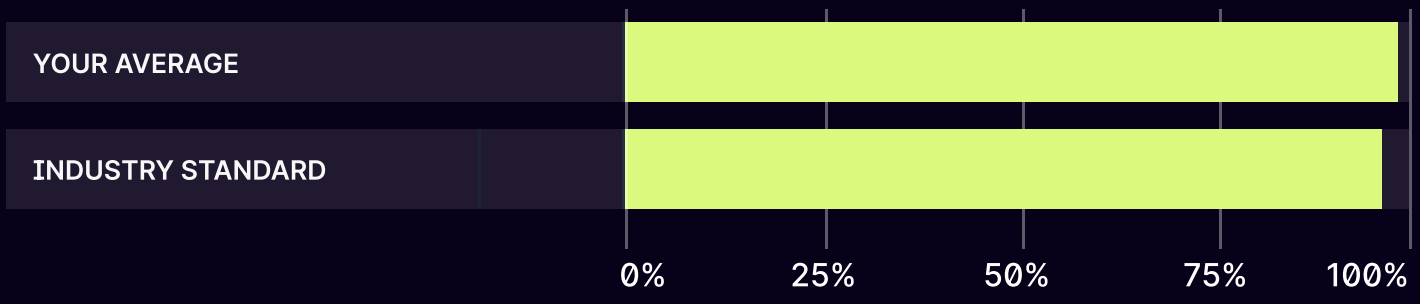
This document outlines the overall security of the Stratis Group Ltd smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Stratis Group Ltd 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



100% 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 Stratis Group Ltd 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 5

Executive Summary 7

Structure and Organization of the Document 8

Complete Analysis 9

Code Coverage and Test Results for all files written by Zokyo Security 17

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Stratis Group Ltd repository:
Repo: - https://github.com/stratisproject/masternode-staking-contract/tree/initial
Last commit: ddc5794e42d49f41b3ad0b5232aa26dee0ced190
- https://github.com/stratisproject/distribution-contract
Last commit: b19932434c7001256e785b38fdd87ea86d3d4831
Fixes: https://github.com/stratisproject/masternode-staking-contract/compare/main...rounding

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

- MasternodeStakingContract.sol
- DistributionContract.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 Stratis Group Ltd 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.

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 Stratis Group Ltd team and the Stratis Group Ltd 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	Possible loss of rewards due to precision loss	Low	Resolved
2	Use OpenZeppelin's nonReentrant modifier	Informational	Unresolved
3	Accounts' left reward balance after claim will always be 0	Informational	Unresolved
4	Variable withdrawingCollateralAmount is not necessary	Informational	Unresolved
5	No fixed solidity version	Informational	Unresolved
6	Missing events for critical functions	Informational	Unresolved

Possible loss of rewards due to precision loss

In MasternodeStakingContract.sol, the method claim rewards calculate the total dividends as follows:

```
if (totalRegistrations > 0) {
            // All categories of registered accounts are treated as having
identical 'staking' amounts for the purposes of dividing up the rewards.
            totalDividends += (amount / totalRegistrations);
lastBalance += amount;
        }
```

Here, there is a `wei` loss of rewards funds due to automatic truncation in the solidity division.

Although a small amount, if repeated multiple times whenever there is a registration, claim rewards, or withdrawal, it will break the invariant that total rewards claimed by users equals the total reward sent to the contract.

PoC:

```
function test_precisionLoss() public {
        // user 1
        registerALegacyAccount();

        // 31 ether as rewards
        vm.deal(
            address(stakingContract),
            address(stakingContract).balance + 31 ether
        );

        // user 2
        registerANonLegacyAccount();

        // 31 ether as rewards
        vm.deal(
            address(stakingContract),
            address(stakingContract).balance + 31 ether
```

```
        );

        // user 3
        address user3 = makeAddr("user3");
        vm.deal(user3, 1_000_000 ether);
        vm.startPrank(user3);
        stakingContract.register{value: 1_000_000 ether}();

        // 31 ether as rewards
        vm.deal(
            address(stakingContract),
            address(stakingContract).balance + 31 ether
        );

        vm.stopPrank();

        uint256 user1BalanceBeforeClaimingRewards = user.balance;
        uint256 user2BalanceBeforeClaimingRewards =
legacyAccounts[0].balance;
        uint256 user3BalanceBeforeClaimingRewards = user3.balance;

        vm.prank(user);
        stakingContract.claimRewards();

        vm.prank(legacyAccounts[0]);
        stakingContract.claimRewards();

        vm.prank(user3);
        stakingContract.claimRewards();

        uint256 user1BalanceAfterClaimingRewards = user.balance;
        uint256 user2BalanceAfterClaimingRewards =
legacyAccounts[0].balance;
        uint256 user3BalanceAfterClaimingRewards = user3.balance;
```

```
        uint totalRewardsClaimedByUsers = user1BalanceAfterClaimingRewards
-
            user1BalanceBeforeClaimingRewards +
            user2BalanceAfterClaimingRewards -
            user2BalanceBeforeClaimingRewards +
            user3BalanceAfterClaimingRewards -
            user3BalanceBeforeClaimingRewards;

        assertEq(totalRewardsClaimedByUsers, 93 ether);
    }
```

Result:

```
        emit log(val: "Error: a == b not satisfied [uint]")
          emit log_named_uint(key: "      Left", val: 92999999999999999999 [9.299e19])
          emit log_named_uint(key: "     Right", val: 93000000000000000000 [9.3e19])
```

We can see that 1 wei is not sent as a reward to users.

Recommendation:

Update the total dividend calculation logic to make this precision loss as low as possible. One of the suggested methods is pointMultiplier as mentioned here under the subtopic `rounding errors`.

INFORMATIONAL-1 | UNRESOLVED

Use OpenZeppelin's nonReentrant modifier

In MasternodeStakingContract.sol, some methods send native tokens to addresses that are not trusted. Although the CEI pattern is followed, it is advised to use OpenZeppelin's ReentrancyGuard as well to ensure more safety.

Recommendation:

Use nonreentrant modifier on claimRewards(...) and completeWithdrawal(...) method.

Accounts' left reward balance after claim will always be 0

In MasternodeStakingContract.sol, the method claimRewards has the following logic:

```
uint256 claimAmount = accounts[msg.sender].balance;

accounts[msg.sender].lastClaimedBlock = block.number;
if (claimAmount == 0) {
    return;
}
accounts[msg.sender].balance -= claimAmount;
```

Here claimAmout is exactly equal to balance but later on, balance is not directly not set 0.

Recommendation:

To save gas, updated as follows:

```
accounts[msg.sender].balance -= claimAmount;
```

With

```
accounts[msg.sender].balance = 0;
```

INFORMATIONAL-3 | UNRESOLVED

Variable withdrawingCollateralAmount is not necessary

In MasternodeStakingContract.sol, the method startWithdrawal initiates a withdrawal and updates the following states:

```
withdrawingCollateralAmount += applicableCollateral;
totalCollateralAmount -= applicableCollateral;
```

Here, these 2 arithmetic operations are not required because:

- When new rewards are added, they will still be accounted correctly as (contract total balance - lastBalance - totalCollateralAmount - registrationOfffset) will be able to handle it.

Recommendation:
State variable withdrawingCollateralAmount can be avoided as state balance can still be managed if it is not explicitly required to check how much balance is currently going to be withdrawn.

INFORMATIONAL-4 | UNRESOLVED

No fixed solidity version

MasternodeStakingContract use the following floating pragma:

```
pragma solidity ^0.8.20;
```

It allows to compile contracts with various versions of the compiler and introduces the risk of using a different version when deploying than during testing.

Recommendation:
Use a specific version of the Solidity compiler.

Missing events for critical functions

In the **MasternodeStakingContract.sol**, there are missing events for critical functions which contain important state changes such as completeWithdrawal() and claimRewards().

Similarly, in the **Distribution.sol**, there are missing events for the claim() function. Emitting events is the best practice secuirtywise as it helps track the system offchain.

Recommendation:
It is advised to add proper events for the same.

	MasternodeStakingContract.sol DistributionContract.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 Stratis Group Ltd 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 Stratis Group Ltd contract/s requirements for details about issuance amounts and how the system handles these.

Running 5 tests for test/DistributionContract.t.sol:DistributionTest
[PASS] test_ShouldClaim() (gas: 47903)
[PASS] test_failToClaimIfThereIsNothingInContractToClaim() (gas: 46575)
[PASS] test_failToClaimIfTimeNotPassedForPayment() (gas: 8652)
[PASS] test_failToDeployContractIfPayAfterTimestampIsInPast() (gas: 43727)
[PASS] test_failToDeployContractIfRecipientIsAddressZero() (gas: 43810)

Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 5.59ms

Running 18 tests for test/MasternodeStakingContract.t.sol:MasternodeStakingContractTest
[PASS] test_ShouldAssignLegacyAccounts() (gas: 38377)
[PASS] test_ShouldClaimRewards() (gas: 200650)
[PASS] test_ShouldCompleteWithdrawal() (gas: 147733)
[PASS] test_ShouldCompleteWithdrawalForALegacyAccount() (gas: 154034)
[PASS] test_ShouldNotAssignAddressZeroAsLegacyAccounts() (gas: 7790)
[PASS] test_ShouldRegisterALegacyAccount() (gas: 130609)
[PASS] test_ShouldRegisterANonLegacyAccount() (gas: 127801)
[PASS] test_ShouldStartWithdrawal() (gas: 117948)
[PASS] test_ShouldStartWithdrawalForALegacyAccount() (gas: 119996)
[PASS] test_failIfTriedToAssignLegacyAccountsMoreThanOnce() (gas: 26626)
[PASS] test_failsToClaimRewardsIfUserIsNotRegistered() (gas: 128983)
[PASS] test_failsToCompleteWithdrawalIfUserIsNotStartedWithdrawal() (gas: 124190)
[PASS] test_failsToCompleteWithdrawalIfWithdrawDelayHasNotPassed() (gas: 116120)
[PASS] test_failsToRegisterALegacyAccountIfAnyOtherAmount() (gas: 24014)
[PASS] test_failsToRegisterANonLegacyAccountIfAnyOtherAmount() (gas: 20963)
[PASS] test_failsToRegisterIfUserIsNotUnregistered() (gas: 132365)
[PASS] test_failsToStartWithdrawalIfUserIsNotRegistered() (gas: 128291)
[FAIL. Reason: assertion failed] test_precisionLoss() (gas: 484552)

Test result: FAILED. 17 passed; 1 failed; 0 skipped; finished in 5.66ms

Ran 2 test suites: 22 tests passed, 1 failed, 0 skipped (23 total tests)

Failing tests:
Encountered 1 failing test in test/
MasternodeStakingContract.t.sol:MasternodeStakingContractTest
[FAIL. Reason: assertion failed] test_precisionLoss() (gas: 484552)

Encountered a total of 1 failing tests, 22 tests succeeded

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% Uncovered Lines
MasternodeStakingContract.sol	100	100	100	100	
DistributionContract.sol	100	100	100	100	
All Files	**100**	**100**	**100**	**100**	

We are grateful for the opportunity to work with the Stratis Group Ltd 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 Stratis Group Ltd team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo