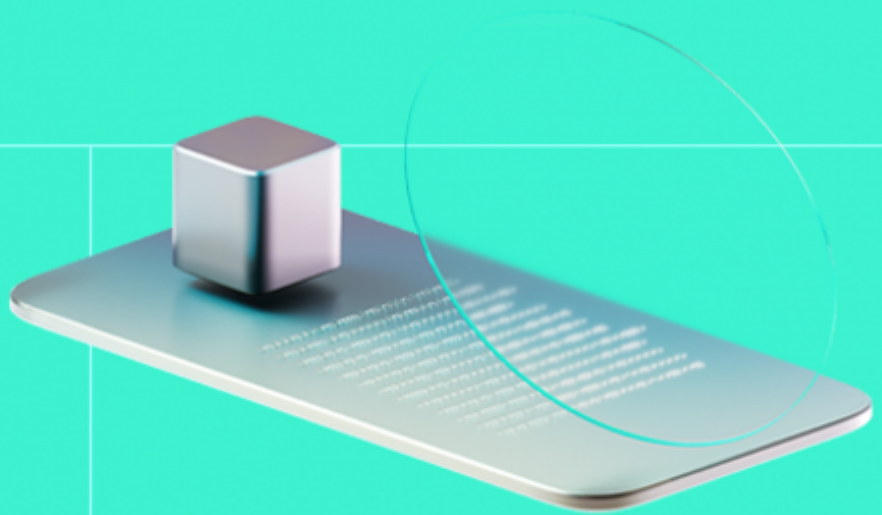




Smart Contract Code Review And Security Analysis Report

Customer: Santiment

Date: 25/03/2024



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

SanrSharesV1 represents a groundbreaking system for traders to monetize their insights by selling access through freely tradable "keys," with key prices tied to their reputation.

Platform: EVM

Language: Solidity

Tags: DEX

Timeline: 08/03/2024 - 11/03/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://gitlab.mind-dev.com/santiopts/sanr-layer-two/
Commit	2022c9e
Remediation Commit	58c87a3

Audit Summary

10/10

Security Score

10/10

Code quality score

100%

Test coverage

10/10

Documentation quality score

Total 10/10

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

5

Total Findings

3

Resolved

2

Accepted

0

Mitigated

Findings by severity

Critical	0
High	1
Medium	0
Low	2

Vulnerability

Status

F-2024-1347 - Missing checks for `address(0)`	Accepted
F-2024-1368 - Missing Reentrancy Modifier	Accepted
F-2024-1343 - Funds Lock Due To Referral Disruption Caused By Malicious Share Address	Fixed
F-2024-1344 - Floating Pragma	Fixed
F-2024-1346 - Fees Are Not Restricted And They Can Be Frontrun	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Santiment
Audited By	Seher Saylik
Approved By	Ataberk Yavuzer
Website	sanr.app
Changelog	12/03/2024 - Preliminary Report
	25/03/2024 - Final Report



Table of Contents

System Overview	6
Privileged Roles	6
Executive Summary	7
Documentation Quality	7
Code Quality	7
Test Coverage	7
Security Score	7
Summary	7
Risks	8
Findings	9
Vulnerability Details	9
Observation Details	18
Disclaimers	24
Appendix 1. Severity Definitions	25
Appendix 2. Scope	26

System Overview

SanR is a marketplace that allows traders selling access to their new signals through “keys” that are freely traded on the market.

SanrSharesV1 - is a contract designed to manage the sale and referral system of shares, incorporating features such as transaction fee settings, ownership management, and referral rewards facilitation. Users can engage in buying and selling shares, benefiting from integrated mechanisms ensuring equitable transactions and fee distributions.

This contract encompasses the following attributes:

- Traders have the capability to utilize previously added share addresses as references, entitling them to earn a percentage from referral fees. To define a share on the platform, a reference is required.
- The platform is structured with three distinct fees:
 1. Protocol fee: Sent to the protocolFeeDestination address specified by the owner, applicable when a user buys or sells a share.
 2. Subject fee: Directed to the share address involved in the transaction.
 3. Referrals fee: Allocated to all referrals associated with the given share address.

Privileged roles

- The owner of the SanrSharesV1 contract can:
 - Set the fee destination address
 - Set the protocol fee percentage
 - Set the subject fee percentage
 - Set the fee percentage for each level of referral
 - Transfer the ownership to another address

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**.

- Functional requirements are partially missed.
- The technical description is not provided.

Code quality

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

- The development environment is configured.
- The code misses some best practices.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Tests were not mandatory due to length of the code.

Security score

Upon auditing, the code was found to contain **0** critical, **1** high, **0** medium, and **2** low severity issues. During the remediation phase, several issues have been addressed, leaving only **1** low severity issue outstanding, 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 **10**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The platform owner holds the authority to adjust the **fees** used in the protocol arbitrarily, potentially **setting them to any value at any time**. Moreover, there is a risk of the cumulative total of these fees reaching or surpassing 100%.
- If the level fee percents do not get provided sequentially, **fees will not be collected from the point where the order is disrupted**. This could lead to inconsistencies in fee collection and potential discrepancies in revenue distribution.
- The price calculated when purchasing shares does not increase proportionally with the supply in the system. In other words, the price increase is not linear. A parabolic pricing logic has been implemented, where the **rate of increase in price grows over time**.
- **Share contract addresses** that represent traders' access to their signals are provided by traders and their code is **out-of-scope**. Therefore, their security cannot be verified by Hacken.
- Due to the utilization of the Ownable contract, there exists a risk whereby the ownership privilege may inadvertently be transferred to an incorrect address due to typographical errors or human errors.

Findings

Vulnerability Details

F-2024-1343 - Funds Lock Due To Referral Disruption Caused By Malicious Share Address - High

Description:

Share addresses progress through a system of references. This means that after the owner defines the initial share address within the protocol, other share contracts can be added by referencing this initial share address. These newly added share contracts can also serve as references for further share contracts. In essence, a chain-like network of references can be established. When subject fees are paid, Ethereum is taken from buyers/sellers and sent from the last reference to the first one sequentially. However, if a maliciously intended referenced share contract stops accepting Ethereum, the fee transfer cannot proceed, causing buy and sell share functions to revert.

In such a scenario, either the funds will remain stuck in the contract, or the system owner may need to reset the referral level fee charges to zero, allowing users to perform transactions without paying referral fees. The affected shares are those created using a malicious referral or a referral that is created using a malicious share address as a referral.

```
function _sendFeeToReferrals(address sharesSubject, uint256 price) p
private {
    address prevReferral = sharesSubject;
    for (uint256 i = 0;; i++) {
        uint256 referralFeePercent = referralLevelsFeePercents[i];
        if (referralFeePercent == 0) break;

        address referral;
        if (prevReferral == owner()) referral = owner();
        else {
            referral = referrals[prevReferral];
            if (referral == address(0)) break;
        }

        earnedByReferralFees[referral] += price * referralFeePercent / 1 ether;
        (bool success1,) = referral.call{value : price * referralFeePercent / 1 ether}("");
        require(success1, "Unable to send referral fee");

        prevReferral = referral;
    }
}
```

The same issue presents for the `protocolFeeDestination` and `sharesSubject` addresses too. If they stop receiving ETH, related all the transactions will revert in the system.

Assets:

- Subscriptions.sol [Not provided]

Status: Fixed

Classification

Severity: High

Impact: Likelihood [1-5]: 3
Impact [1-5]: 5
Exploitability [1-2]: 1
Complexity [0-2]: 1
Final Score: 3.8 [HIGH]

Recommendations

Recommendation: Instead of sending fees to referrals, subject address and to protocol fee destination directly in the `buyShares()` and `sellShares()` functions, implement pull-over-push pattern to allow them to withdraw the accumulated fee balance from the contract.

Remediation (Revised commit: 960c748): The Santiment team, has adopted a pull-over-push model for all payments made to the share subject, protocol fee or referrals by recording native assets to the balance mapping instead of directly transferring them via the call function, enabling them to withdraw from the contract later.

Evidences

PoC

Reproduce:

Steps to Reproduce:

Initiating 4 share subjects:

- The system owner creates the first share subject and executes `buyFirstShare()` function.
- A second share subject is created in the platform by giving the first share address as a referral.
- A second third share subject is created in the platform by giving the second share address as a referral by a malicious address.
- A fourth share subject is created by using the malicious subject address as a referral.

Owner initiates fees:

- The system owner sets the required fees including referral fee level percents for each level of share.

Users buying different shares:

- Users are buying from the fourth share and the fees are sent to the share creator and to all the referrals including first, second and third share subjects.

Malicious share subject stops receiving Eth:

- The malicious share address activates the reverting fallback function in its implementation to block any possible Eth transfer to it.

Impact on Subsequent Transactions:

- Due to this block, holders of the fourth share are no longer able to sell the tokens as the share fee transfers in `_sendFeeToReferrals()` will always fail when the portion for the third(malicious) contract address is tried to be sent.

Results:

Foundry tests:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/console.sol";

import {Test, console2} from "forge-std/Test.sol";
import {SanrSharesV1} from "../src/Subscriptions.sol";
import {MaliciousShareContract} from "../src/MaliciousShareContract.sol";

contract SanrShares is Test {
    address protocolFeeDestination = vm.addr(1);
    address sharesSubject = vm.addr(2);
    address user1 = vm.addr(3);
    address user2 = vm.addr(4);
    address malicious = vm.addr(5);

    SanrSharesV1 sanrShares;
    MaliciousBuyer maliciousBuyer;
    MaliciousShareContract maliciousShare2;

    function setUp() public {
        vm.startPrank(sharesSubject);
        sanrShares = new SanrSharesV1();
        sanrShares.setFeeDestination(protocolFeeDestination);
        sanrShares.buyFirstShare(sharesSubject, 1);
    }

    function test_funds_lock_if_the_referral_doesnt_accept_eth() public {
        address referredShares1= vm.addr(6);
        maliciousShare2 = new MaliciousShareContract(address(sanrShares));
        address referredShares3= vm.addr(8);
        vm.startPrank(referredShares1);
        // New share address is created by using the previous share as referral
        sanrShares.buyFirstShare(sharesSubject, 1);

        // New share address is created by using the previous share as referral
        maliciousShare2.buyFirstShare(referredShares1, 1);

        vm.startPrank(referredShares3);
        // New share address is created by using the previous share as referral
        sanrShares.buyFirstShare(address(maliciousShare2), 1);

        vm.startPrank(sharesSubject);
```

```
// protocol fee = 10%
sanrShares.setProtocolFeePercent(0.1 ether);
// subject fee = 5%
sanrShares.setSubjectFeePercent(0.05 ether);
// fee percents and levels
sanrShares.setReferralLevelFeePercent(0.05 ether, 0);
sanrShares.setReferr
```

[See more](#)

F-2024-1346 - Fees Are Not Restricted And They Can Be Frontrunned

- Low

Description:

The fees utilized within the contract—protocol fees, subject fees, and referral fees—are not limited. The platform owner retains the authority to adjust these fees at any time and to any value. Furthermore, their cumulative total may reach or even exceed 100%. Consequently, users may find themselves in a scenario where their entire transaction amount is consumed by fees, resulting in minimal or no returns when they are executing the sell function.

As mentioned above, the owner can submit a transaction to alter the fees concurrently. By allocating a higher gas limit to their transaction, the owner can ensure it is processed faster, resulting in the user paying a different amount of fees than anticipated. This situation allows the owner to engage in front-running, exploiting their ability to preempt user transactions and alter exchange outcomes to their advantage.

```
//for owner
function setFeeDestination(address _feeDestination) public onlyOwner
{
    protocolFeeDestination = _feeDestination;
}
function setProtocolFeePercent(uint256 _feePercent) public onlyOwner
{
    protocolFeePercent = _feePercent;
}
function setSubjectFeePercent(uint256 _feePercent) public onlyOwner
{
    subjectFeePercent = _feePercent;
}
function setReferralLevelFeePercent(uint256 _feePercent, uint256 _level) public onlyOwner {
    referralLevelsFeePercents[_level] = _feePercent;
}
```

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Classification

Severity:

Low

Impact:

Likelihood [1-5]: 3

Impact [1-5]: 3

Exploitability [1-2]: 2

Complexity [0-2]: 0

Final Score: 2.080083823051904 [LOW]

Recommendations

Recommendation: To mitigate the risk of excessive fees consuming users' transaction amounts and potentially leaving them with minimal returns, it is advisable to implement fee caps or limits within the contract. Additionally, to address frontrunning issues, make the contract pausable and allow fee adjustments only when the contract is in a paused state.

Remediation (Revised commit: 58c87a3): The Santiment team integrated check statements into the fee-setting functions to ensure that each fee type—protocol fee, subject fee, and the cumulative sum of referral level fees—is individually capped at 0.1 Ether.

[F-2024-1347](#) - Missing checks for `address(0)` - Low

Description:

In Solidity, the Ethereum address

`0x0000000000000000000000000000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "

Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.

For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

Affected functions:

- `setFeeDestination()`

Assets:

- Subscriptions.sol [Not provided]

Status:

Accepted

Classification

Severity:

Low

Impact:

Likelihood [1-5]: 2

Impact [1-5]: 2

Exploitability [1-2]: 1

Complexity [0-2]: 0

Final Score: 2.0 (Low)

Recommendations

Recommendation:

It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Remediation (Accepted): The Santiment team acknowledged the given finding.

[F-2024-1344](#) - Floating Pragma - Info

Description:

The project uses floating pragmas $\geq 0.8.2 < 0.9.0$

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Classification

Severity:

Info

Recommendations

Recommendation:

Fix the version to a new one and remove the floating pragma.

Remediation (Revised commit: d991114): The Santiment team removed the floating pragma and fixed the Solidity version to 0.8.23.

[F-2024-1368](#) - Missing Reentrancy Modifier - Info

Description:

A reentrancy attack is a common vulnerability in smart contracts, particularly when a contract makes an external call to another untrusted contract and then continues to execute code afterwards. If the called contract is malicious, it can call back into the original contract in a way that causes the original function to run again, potentially leading to effects like multiple withdrawals and other unintended actions.

The absence of reentrancy guards, such as the **nonReentrant** modifier provided by OpenZeppelin's ReentrancyGuard contract, makes a function susceptible to reentrancy attacks. This modifier prevents a function from being called again until it has finished executing.

The functions **buyFirstShare()**, **buyShares()** and **sellShares()** functions lack proper reentrancy locks.

Assets:

- Subscriptions.sol [Not provided]

Status:

Accepted

Classification

Severity:

Info

Recommendations

Recommendation:

Consider adding **nonReentrant** modifier to given functions.

Remediation (Accepted): The Santiment team acknowledged this finding and decided not addressing it, citing concerns regarding Gas consumption. This decision was made considering that the current implementation doesn't present any reentrancy vulnerabilities, as the Checks Effects Interactions (CEI) pattern is adhered to correctly.

Observation Details

F-2024-1345 - Missing Event Emitting - Info

Description:

Events for critical state changes should be emitted for tracking actions off-chain.

Events are crucial for tracking changes on the blockchain, especially for actions that alter significant contract states or permissions. The absence of events in these functions means that external entities, such as user interfaces or off-chain monitoring systems, cannot effectively track these important changes.

It was observed that events are missing events in the following functions:

- `setFeeDestination()` ,
- `setProtocolFeePercent()` ,
- `setSubjectFeePercent()` ,
- `setReferralLevelFeePercent()`

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Recommendations

Recommendation:

Consider events emitting on aforementioned functions.

Remediation (Revised commit: 4607271): The Santiment team added the required events for the critical state updates.

[F-2024-1381](#) - Missing Contract Existence Checks Before Low-level Calls - Info

Description:

Low-level calls return success if there is no code present at the specified address. In addition to the zero-address checks, add a check to verify that **<address>.code.length > 0**.

Otherwise, the contract may make calls to those zero addresses and cause burning the transferred Eth amount.

There are 5 instances of this issue:

```
./src/Subscriptions.sol
341: (bool success0,) = msg.sender.call{value : msg.value - totalPrice}("");
348: (bool success1,) = protocolFeeDestination.call{value : protocolFee}("");
350: (bool success2,) = sharesSubject.call{value : subjectFee}("");
368: (bool success1,) = msg.sender.call{value : price - protocolFee - subjectFee - referralsFee}("");
369: (bool success2,) = protocolFeeDestination.call{value : protocolFee}("");
371: (bool success3,) = sharesSubject.call{value : subjectFee}("");
```

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Recommendations

Recommendation:

To enhance the security and reliability of your Solidity smart contracts, always include contract existence checks before making low-level calls. In addition to verifying that the address is not the zero address, also confirm that **<address>.code.length > 0**. These checks help ensure that the target address corresponds to a valid and functioning contract, reducing the risk of unexpected behavior and vulnerabilities in your contract interactions.

Remediation (Revised commit: 960c748): The Santiment team has removed the low-level calls from `buyShares()` and `sellShares()` function.s

[F-2024-1384](#) - External Calls In An Unbounded Loop Can Result In A DoS - Info

Description:

In the platform, a share subject may be associated with several referrals. In each transaction, whether it involves buying or selling, a fee is distributed among these referrals. However, should the size of this list become sufficiently large, it could potentially lead to a denial of service issue, hindering the execution of the `_sendFeeToReferrals()` function. In blockchains, transactions have Gas limits and large iterations of loops may exceed this limit. Consequently, all transactions pertaining to the relevant share subject would encounter disruption.

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Recommendations

Recommendation:

To mitigate the risk of Denial-of-Service (DoS) attacks in your Solidity code, it's important to limit the number of iterations in loops that involve external calls. A single failed external call in an unbounded loop can lead to a revert, causing disruptions in contract execution. Consider implementing safeguards, such as setting a maximum level for referrer to be.

Remediation (Revised commit: `f1fae89`): The Santiment team limited the number of total referral levels to 50 to prevent possible future DoS risks..

[F-2024-1392](#) - `msg.value` and `msg.sender` Should Be Cached - Info

Description: Cacheing saves gas when compared to repeating the calculation at each point it is used in the contract.

msg.sender and **msg.value** are used several times without cacheing them in the **buyShares()** and **sellShares()** functions.

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Recommendations

Recommendation: Consider caching **msg.value** and **msg.sender** in a local variable at the start of payable functions where it is accessed multiple times. Implement this optimization by assigning **msg.value** and **msg.sender** to local variables and using these variables for subsequent operations within the function.

For example:

```
function myPayableFunction() public payable {  
    uint256 cachedValue = msg.value; // Use cachedValue instead of msg.v  
    alue for further operations  
}
```

Remediation (Revised commit: 89c0f4e): The msg.value and msg.sender values are cached,

[F-2024-1394](#) - Revert String Size Optimization - Info

Description:

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead to calculate memory offset, etc.

```
./src/Subscriptions.sol
102: require(newOwner != address(0), "Ownable: new owner is the zero address"); // @audit-issue: String length is `38`
327: require(sharesSupply[sharesSubject] > 0, "First share should be purchased by buyFirstShare"); // @audit-issue: String length is `48`
```

Status:

Fixed

Recommendations

Recommendation:

To optimize gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of gas used during deployment and runtime when the revert condition is met.

Remediation (Revised commit: b129f559): The Santiment team shortened the revert string sizes.

F-2024-1460 - Owner Can Renounce Ownership - Info

Description:

The smart contract under inspection inherits from the **Ownable** library, which provides basic authorization control functions, simplifying the implementation of user permissions. While the contract allows for the transfer of ownership to a different address or account, it also retains the default **renounceOwnership** function from **Ownable**. Once the owner uses this function to renounce ownership, the contract becomes ownerless. Evidence in the transaction logs shows that, following the activation of the **renounceOwnership** function, any attempts to invoke functions requiring owner permissions fail, with the error message: **"Ownable: caller is not the owner."** This condition makes the contract's adjustable parameters immutable, potentially rendering the contract ineffective for any future administrative modifications that might be needed.

[Ownable2Step](#) prevents the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

Assets:

- Subscriptions.sol [Not provided]

Status:

Fixed

Recommendations

Recommendation:

Consider using **Ownable2Step** instead of **Ownable** from OpenZeppelin Contracts to enhance the security of your contract ownership management. This contract prevents the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call. This two-step ownership transfer process adds an additional layer of security to your contract's ownership management.

Remediation (Revised commit: 6474c1a): The Santiment team removed the **renounceOwnership()** function from the contract.

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](https://github.com/hacken/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://gitlab.mind-dev.com/santiopts/sanr-layer-two/
Commit	2022c9e
Whitepaper	N/A
Requirements	N/A
Technical Requirements	N/A

Contracts in Scope

Subscriptions.sol