# Security Review Report
# NM-0194 STARKNET TOKEN DISTRIBUTOR

NETHERMIND SECURITY

(Feb 22, 2024)

## Contents

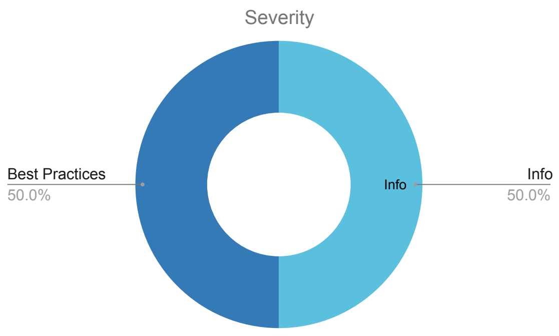
# 1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the StarkNet Foundation on the **Token Distributor Contract** containing 108 lines of code. The distributor contract is designed to distribute tokens utilizing Merkle tree verification. A Merkle tree is a binary tree where leaf nodes represent data, internal nodes represent hashed pairs of child nodes, and the root node represents the total hash of the data set. The tree is managed off-chain, with only the root hash being published to the contract by the owner. Multiple roots can be published, each representing the tree's state at a different point in time.

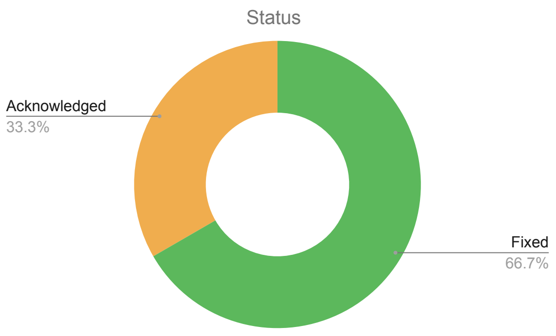
When users wish to make a claim, they need to query the off-chain service for their claim data, including the amount and proofs for on-chain validation then submit it to the distributor contract. The contract uses this data to calculate the Merkle tree root and then compares it with the published roots in storage. If any one of the roots matches, the claim is valid, and the contract sends the token to the user.

**The audit was performed using**: (a) manual analysis of the codebase, (b) simulation of the smart contracts and (c) automation analysis of the smart contracts. **Along this document, we report** 6 points of attention, where three are classified as `Info`, and three are classified as `Best Practice`. The issues are summarized in Fig. 1.

**This document is organized as follows.** Section 2 presents the files in the scope of this audit. Section 3 presents the assumptions for this audit. Section 4 summarizes the issues. Section 5 presents the system overview. Section 6 discusses the risk rating methodology adopted for this audit. Section 7 details the issues. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, and automated tests. Section 10 concludes the document.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

**Fig 1: (a) Distribution of issues: Critical** (0), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (3), **Best Practices** (3). **(b) Distribution of status: Fixed** (4), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

**Summary of the Audit**

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | Feb 19, 2024 |
| **Final Report** | Feb 22, 2024 |
| **Methods** | Manual Review, Automated analysis |
| **Repository** | defispring |
| **Commit Hash** | 5ca2536c6cfccb7527dd633796d46a0d2f74febd |
| **Final Commit Hash** | be3ca1f2b55e1de94399c29da854adb30184bebc |
| **Documentation** | README.md |
| **Documentation Assessment** | High |
| **Test Suite Assessment** | High |

## 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | src/contract.cairo | 106 | 10 | 9.4% | 22 | 138 |
| 2 | src/lib.cairo | 2 | 0 | 0.0% | 0 | 2 |
| | **Total** | **108** | **10** | **9.3%** | **22** | **140** |

The class hash of the Distributor contract is 0x006a54af2934978ac59b27b91291d3da634f161fd5f22a2993da425893c44c64.

## 3 Assumptions

The prepared security review is based on the following assumptions:

- The off-chain code generating the root of the Merkle tree is correctly implemented.
- The Alexandria library accurately implements the Merkle tree.
- The contract owner always provides enough tokens and doesn't act maliciously. In other words, the owner is trusted.
- The STRK token to be distributed is compliant with and conforms to the ERC20 token standard.
- The amount that users can claim in each round increases monotonically.

## 4 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Hardcoded token address | Info | Acknowledged |
| 2 | Low trust on Alexandria Merkle Tree library | Info | Acknowledged |
| 3 | Possible hash collision when calculating the hash of leaves | Info | Fixed |
| 4 | Maintainable imports rather single line for every imports | Best Practices | Fixed |
| 5 | Unchecked return value for token transfer | Best Practices | Fixed |
| 6 | Use of Poisedon hashing instead of Pedersen | Best Practices | Fixed |

## 5 System Overview

The **Distributor** contract facilitates the distribution of the STRK token to eligible participants through the use of a Merkle tree.

A Merkle tree is a binary tree where leaf nodes represent data, internal nodes represent hashed pairs of child nodes, and the root node represents the total hash of the data set. The tree is managed off-chain, with only the root hash being published to the contract by the owner. Multiple roots can be published, each representing the tree's state at a different point in time.

When users wish to make a claim, they need to query the off-chain service for their claim data, including the amount and proofs for on-chain validation then submit it to the distributor contract. The contract uses this data to calculate the Merkle tree root and then compares it with the published roots in storage. If any one of the roots matches, the claim is valid, and the contract sends the token to the user.

This contract comprises a concise set of functions:

1. `claim(amount: u128, proof: Span::<felt252>)`: This enables users to claim a specified amount of tokens by providing a valid Merkle proof.
2. `add_root(new_root: felt252)`: This is reserved for the contract owner to add a new Merkle root, thereby specifying eligibility for token claims.
3. `get_root_for(claimee: ContractAddress, amount: u128, proof: Span::<felt252>)`: This retrieves the Merkle root associated with a particular user's claim, along with the specified amount and corresponding Merkle proof.
4. `amount_already_claimed(claimee: ContractAddress)`: This returns the amount of tokens already claimed by a specific user.
5. `roots()`: This returns the list of all Merkle roots currently registered in the contract.

The contract owner is responsible for adding Merkle roots by calling the `add_root(...)` function, with each root representing eligibility for token claims for individual users.

Users can claim their tokens using the `claim(...)` function along with valid Merkle proofs. Upon a successful claim, the `allocation_claimed` mapping is updated to prevent duplicate claims, ensuring each user can claim tokens only once per Merkle root.

# 6 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

**Likelihood** measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

**Impact** is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

| | | **Severity Risk** | | |
|---|---|---|---|---|
| **Impact** | **High** | Medium | High | Critical |
| | **Medium** | Low | Medium | High |
| | **Low** | Info/Best Practices | Low | Medium |
| | **Undetermined** | Undetermined | Undetermined | Undetermined |
| | | **Low** | **Medium** | **High** |
| | | **Likelihood** | | |

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

# 7 Issues

## 7.1 [Info] Hardcoded token address

**File(s)**: `contract.cairo`

**Description**: The $STRK address is currently hardcoded to its address on the Sepolia testnet. This address may differ on the mainnet, which would require manual checks and code changes before deployment.

**Recommendation(s)**: Consider injecting the $STRK address via the constructor.

**Status**: Acknowledged

**Update from the client**: We have verified that the mainnet address will not differ from the testnet addresses. We want to avoid complicating the constructor to minimize room for error during deployment of the contract.

## 7.2 [Info] Low trust on Alexandria Merkle Tree library

**File(s)**: `contract.cairo`

**Description**: In the current setup, there exists an external dependency originating from the Alexandria library. This particular external dependency has not undergone an audit, which raises concerns about the overall integrity and reliability of the system.

The following code snippet is where the concern arises:

```
use alexandria_merkle_tree::merkle_tree::{
    Hasher, MerkleTree, pedersen::PedersenHasherImpl, MerkleTreeTrait
};
```

**Recommendation(s)**: Consider monitoring this library for potential future issues. This can be done by regularly checking updates and changes made to the library. Also, consider conducting an audit of the library to ensure its integrity and reliability.

**Status**: Acknowledged

**Update from the client**: Acknowledged. We will continue monitoring the library for potential future issues.

## 7.3 [Info] Possible hash collision when calculating the hash of leaves

**File(s)**: `contract.cairo`

**Description**: Presently, the leaf of the Merkle tree is computed using the following code snippet:

```
let leaf = LegacyHash::hash(claimee.into(), amount);
merkle_tree.compute_root(leaf, proof)
```

The function `LegacyHash::hash()` preprocesses the input values into `felt252` before hashing. Consequently, the output of this function represents the Pedersen hash of two `felt252` values. Similarly, the middle node in the Merkle tree also corresponds to the Pedersen hash of two `felt252` values. Therefore, in theory, it is conceivable that an attacker could provide `claimee` as the left middle node and `amount` as the right middle node, leading to a situation where a valid claim could be made to drain the contract. However, this attack vector is contingent upon two conditions:

- The value of the right middle node must fit within `u128`, as it is provided as the input parameter `amount`;
- The left middle node corresponds to the caller's address. While practically improbable, it's worth noting that in the future, the contract might introduce a function to allow claims on behalf of others;

**Recommendation(s)**: Consider avoiding using leaf values that are two `felt252` prior to hashing or using a different hashing function other than Pedersen for hashing leaves.

**Status**: Fixed

**Update from the client**: Fixed in commit `5f06c147b42daf3685d3cb9a5f58c67edc757998` on branch audit-fixes by hashing leaves twice: `leaf = LegacyHash::hash(LegacyHash::hash(claimee.into(), amount), 1);`

**Update from Nethermind**: Hashing has changed, but the test data has not been updated. As a result, the test cases are failing.

**Update from the client**: Fixed in commit `be3ca1f2b55e1de94399c29da854adb30184bebc` on branch audit-fixes by using Poseidon for hashing the leaves.

## 7.4 [Best Practices] Maintainable imports rather single line for every imports

**File(s)**: `contract.cairo`

**Description**: In the current coding style of the contract, individual external libraries are imported one by one, each on a single line. This approach, while functional, does not align with best coding practices. It is generally advised to consolidate these imports into a single line, where possible, to improve the readability and maintainability of the code.

To illustrate, here is a snapshot of how the external libraries are currently imported into the contract:

```
use openzeppelin::access::ownable::ownable::OwnableComponent::InternalTrait;
use openzeppelin::access::ownable::ownable::OwnableComponent;
```

**Recommendation(s)**: Instead of having multiple lines for each import, it would be more advantageous and in line with best practices to group them into a single line. This would not only make the code cleaner but also easier to manage and maintain.

The following is an example of how the imports can be grouped:

```
use openzeppelin::access::ownable::ownable::{OwnableComponent::InternalTrait,OwnableComponent}; // @audit - Best
↪  Practice
```

**Status**: Fixed

**Update from the client**: Fixed in commit `c91d05feb3b5b7ca06bce3593144021bac2ebad9on` branch audit-fixes.

## 7.5 [Best Practices] Unchecked return value for token transfer

**File(s)**: `contract.cairo`

**Description**: It is usually good to check the return value unless one is certain the given token will revert in the event of a failure. In this instance, the $STRK token is currently set to the testnet address, and the code can also be upgraded in the future.

```
token.transfer(claimee, u256 { high: 0, low: left_to_claim });
```

**Recommendation(s)**: Consider checking the return value of the token transfer call.

**Status**: Fixed

**Update from the client**: Fixed in commit `a20d0be7dc4ab5441770c3cd9a190a9444f1bb3f` on branch audit-fixes

## 7.6 [Best Practices] Use of Poisedon hashing instead of Pedersen

**File(s)**: `contract.cairo`

**Description**: The current hashing technique implemented in this contract is Pedersen hashing. Pedersen hashing, while effective in certain use cases, can be resource-intensive and may not always be the optimal choice. It is recommended to switch to Poseidon hashing, a different hashing algorithm that is known to be significantly less costly in terms of computational resources. This shift in technique can potentially improve the overall efficiency of the contract.

**Recommendation(s)**: It is recommended to use Poseidon hashing as it is much more gas-efficient compared to Pedersen hashing.

**Status**: Fixed

**Update from the client**: Fixed in commit `5f06c147b42daf3685d3cb9a5f58c67edc757998` on branch audit-fixes

**Update from Nethermind**: Commit 5f06c147 changed the hashing of both leaves and middle nodes to Poseidon in the backend side. However, on the contract side, only middle nodes are calculated using Poseidon, while leaves are still calculated using Pedersen ( `Legacy::hash()` uses Pedersen as can be seen here).

**Update from the client**: Acknowledged. We will stick with Pedersen hashing for the tree because the backend is already stabilized, but use Poseidon for the leaves.

# 8 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

– Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;

– User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;

– Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;

– API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;

– Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;

– Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

**Remarks about documentation**

The `StarkNet Foundation` has provided documentation about the distributor contract in the codebase comments and through the README file in the repository. The provided documentation is sufficient for the audit.

# 9 Test Suite Evaluation

## 9.1 Compilation Output

```
> scarb build
Compiling distributor v0.1.0 (/home/audits/NM-0194/contract/Scarb.toml)
    Finished release target(s) in 4 seconds
```

## 9.2 Tests Output

```
> snforge test
Compiling distributor v0.1.0 (/home/audits/NM-0194/contract/Scarb.toml)
    Finished release target(s) in 5 seconds


Collected 5 test(s) from distributor package
Running 5 test(s) from src/
[PASS] distributor::tests::test::test_compute_root (gas: ~2213)
[PASS] distributor::tests::test::test_claim_wrong_claimee (gas: ~3319)
[PASS] distributor::tests::test::test_claim_wrong_amount (gas: ~3319)
[PASS] distributor::tests::test::test_claim_invalid_proof (gas: ~9080)
[PASS] distributor::tests::test::test_single_claims_multiple_roots (gas: ~20230)
Tests: 5 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

# 10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

**Blockchain Security:** At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

**Blockchain Core Development:** Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

**DevOps and Infrastructure Management:** Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

**Cryptography Research:** At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

**Smart Contract Development & DeFi Research:** Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

**Our suite of L2 tooling:** Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

**Learn more about us at nethermind.io.**

### General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

### Disclaimer