



QuillAudits

# Audit Report November, 2021

For



# Contents

Scope of Audit	01
Check Vulnerabilities	01
Techniques and Methods	02
Issue Categories	03
Number of security issues per severity.	03
Introduction	04
Issues Found – Code Review / Manual Testing	05
High Severity Issues	05
Medium Severity Issues	05
Low Severity Issues	05
1. Outdated Compiler Version (SWC 102)	05
2. Public function that could be declared external	06
3. Costly loop	07
4. Using of approve function of token standard	08
Informational Issues	09
Functional test	10
Results	11
Closing Summary	12



## Scope of the Audit

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

## Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20/721 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level



## Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20/721 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

### Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

### Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

### Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

### Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

### Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis, Theo.



## Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

## Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	4	3
Closed	0	0	0	0

## Introduction

During the period of **9th Nov 2021 to 16th Nov 2021** - QuillAudits Team performed a security audit for **Nexity** smart contracts.

The code for the audit was taken from the following official link:  
[https://etherscan.io/  
address/0xebeef419bb5a347e3d98f7d2168055214d12cbdb#code](https://etherscan.io/address/0xebeef419bb5a347e3d98f7d2168055214d12cbdb#code)





## Issues Found

### A. Contract – TaxableTeamToken

#### High severity issues

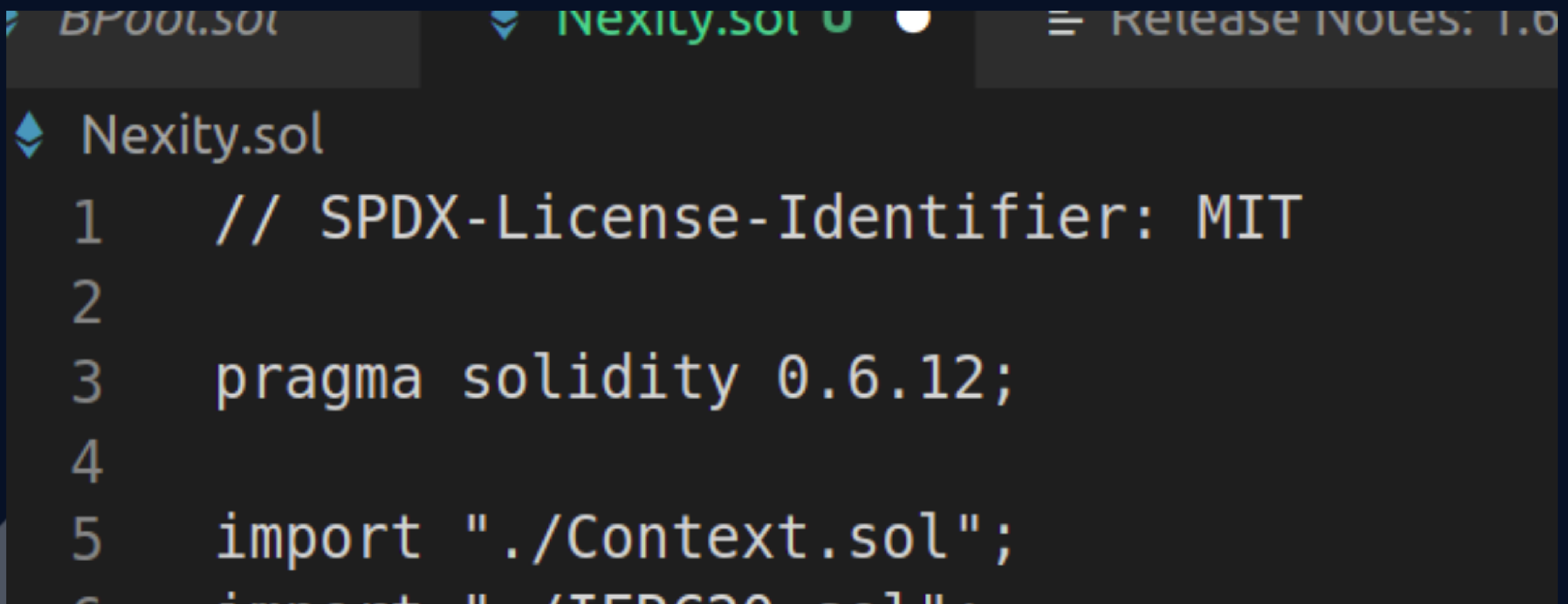
No issues were found.

#### Medium severity issues

No issues were found.

#### Low level severity issues

##### 1. Outdated Compiler Version (SWC 102) → All solidity files



```

Nexity.sol
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity 0.6.12;
4
5  import "./Context.sol";
6  import "./ERC20.sol";

```

#### Description

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

#### Remediation

It is recommended to use a recent version of the Solidity compiler, which is [Version 0.8.9](#)

Status: **Acknowledged**

## 2. Public function that could be declared external

### Description: In TaxableTeamToken.sol

A function with a **public** visibility modifier that is not called internally. Changing the visibility level to **external** increases code readability. Moreover, in many cases, functions with **external** visibility modifiers spend less gas compared to functions with public visibility modifiers.

The functions defined in the file, which are marked as **public**, are below

- reflect
- "decreaseAllowance"
- increaseAllowance
- reflectionFromToken

However, it is never directly called by another function in the same contract or in any of its descendants. Consider marking it as "external" instead.

### Recommendations

Use the external functions never called from the contract via internal call. Reading [Link](#).

Status: **Acknowledged**



### 3. Costly Loop In TaxableTeamToken.sol

#### Description

The loop in the contract includes state variables like the `.length` of a non-memory array in the condition of the for loops.

As a result, these state variables consume a lot more extra gas for every iteration of the for a loop.

The below functions include such loops at the above-mentioned lines:

- `includeAccount`
- `_getCurrentSupply`

#### Recommendations

It's quite effective to use a local variable instead of a state variable like `.length` in a loop.

For instance,

```
uint256 local_variable = _groupInfo.addresses.length;
for (uint256 i = 0; i < local_variable; i++) {
    if (_groupInfo.addresses[i] == msg.sender) {
        _isAddressExistInGroup = true;
        _senderIndex = i;
        break;
    }
}
```

Reading reference link: <https://blog.b9lab.com/getting-loopy-with-solidity-1d51794622ad>

Status: **Acknowledged**

## 4. Using the approve function of the token standard

### Description: In TaxableTeamToken.sol

The approve function of ERC-20 is vulnerable. Using a front-running attack, one can spend approved tokens before the change of allowance value.

```

109
110     function approve(address spender, uint256 amount) public override returns (bool) {
111         ... _approve(_msgSender(), spender, amount);
112         ... return true;
113     }
114

```

### Recommendation

To prevent attack vectors described above, clients should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. Though the contract itself shouldn't enforce it, to allow backward compatibility with contracts deployed before.

Detailed reading around it can be found at [EIP 20](#)

Status: **Acknowledged**



## Informational issues

1. Reentrancy Guard missing: Use the reentrancy OpenZeppelin guard to avoid improper Enforcement of Behavioral Workflow. [Link1](#), [Link2](#).

**Status:** Acknowledged

2. Upgradeable contract was missing. Smart contracts deployed using OpenZeppelin Upgrades Plugins can be upgraded to modify their code while preserving their address, state, and balance. This allows you to iteratively add new features to your project, or fix any bugs you may find in production. [Link](#). Later after a discussion with the client, they suggested switching to a new contract isn't in their plan. So we acknowledged this issue.

**Status:** Acknowledged

3. Linting issues were found

```
contract.sol
117:2  error  Line length must be no more than 120 but current length is 130  max-line-length
127:2  error  Line length must be no more than 120 but current length is 138  max-line-length
217:2  error  Line length must be no more than 120 but current length is 126  max-line-length
225:2  error  Line length must be no more than 120 but current length is 126  max-line-length
234:2  error  Line length must be no more than 120 but current length is 126  max-line-length
243:2  error  Line length must be no more than 120 but current length is 126  max-line-length
270:2  error  Line length must be no more than 120 but current length is 127  max-line-length

✖ 7 problems (7 errors, 0 warnings)
```

**Status:** Acknowledged

## Functional test

Function Names	Testing results	Testing results
approve()	Passed	Passed
transferFrom	Passed	Passed
increaseAllowance	Passed	Passed
decreaseAllowance	Passed	Passed
reflect	Passed	Passed
reflectionFromToken	Passed	Passed
tokenFromReflection	Passed	Passed
excludeAccount	Passed	Passed
setFeesPercentage	Passed	Passed
_transfer	Passed	Passed
_transferStandard	Passed	Passed
_transferToExcluded	Passed	Passed
_transferFromExcluded	Passed	Passed
_transferBothExcluded	Passed	Passed
_reflectFee	Passed	Passed
_getValues	Passed	Passed
_getTValues	Passed	Passed
_getRValues	Passed	Passed
_getRate()	Passed	Passed
_getCurrentSupply()	Passed	Passed



## Results

Few issues were highlighted in the issue section. Some false positive errors were reported by the tool. All the other issues have been categorized above according to their level of severity, which has been acknowledged by Nexity Network Team.



## Closing Summary

Overall, smart contracts are very well written and adhere to guidelines.

No instances of Integer Overflow and Underflow vulnerabilities or Back-Door Entry were found in the contract, but relying on other contracts might cause Reentrancy Vulnerability.



## Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of **Nexity Network**. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough; we recommend that the **Nexity** Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.





# Audit Report November, 2021

For



QuillAudits

📍 Canada, India, Singapore, United Kingdom

🌐 [audits.quillhash.com](https://audits.quillhash.com)

✉️ [audits@quillhash.com](mailto:audits@quillhash.com)