



INFINITY



BLOCKCHAIN SOLUTIONS

Smart contract Audit
Full Detailed Report





Table of contents

Table of Contents

Introduction	4
Project Background	4
Audit scope	4
Claimed Smart Contract Features.....	5
Audit Summary	6
Risk Analysis	7
Documentation	8
Use of Dependencies	8
AS-IS overview	9
Severity Definitions	10
Audit Findings	11
Critical Severity.....	11
High Severity	11
Medium.....	11
Low	11
Very Low / Informational / Best practices:	11
Centralization.....	12
Conclusion.....	13
Our Methodology.....	14
Disclaimers.....	16
Infinity Blockchain Solutions.io Disclaimer.....	16
Technical Disclaimer	16
Appendix.....	17
Slither Results Log.....	18
Solidity Static Analysis	19
Solhint Linter.....	22

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

Infinity Blockchain Solutions was contracted by the Neo team to perform the Security audit of the Neo BlockChain Bank smart contract code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on January 19th, 2024.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

- The NBCB is an ERC20-based standard smart contract deployed on the Polygon blockchain.

Audit scope

Name	Code Review and Security Analysis Report for Neo Blockchain Bank Smart Contract
Platform	Polygon
Language	Solidity
File	Neo.sol
Polygon Code	0x62597B43DBEfCCb97A0c6a8efbc8A451e4828254
Audit Date	January 19th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
<p>Tokenomics:</p> <ul style="list-style-type: none">• Name: Neo BlockChain Bank• Symbol: NBCB• Decimals: 18• Total Supply: 9,999,999,000	<p>YES, This is valid.</p>
<p>Ownership control:</p> <ul style="list-style-type: none">• Current owner can transfer the ownership.• Owner can renounce ownership.	<p>YES, This is valid. We advise renouncing ownership to make the contract fully decentralized.</p>

Audit Summary

According to the standard audit assessment, Customer`s solidity based smart contracts are **“Secured”**. **Also, these contracts contain owner control, which does not make them fully decentralized.**

We used various tools like Slither, Solhint and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section.

We found 0 critical, 0 high, 0 medium and 0 low and 0 very low-level issues.

Investors Advice: Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Risk Analysis

Category	Result
● Buy Tax	None
● Sell Tax	None
● Cannot Buy	False
● Cannot Sell	False
● Max Tax	None
● Modify Tax	None
● Fee Check	N/A
● Is Honeypot	Not Detected
● Trading Cooldown	Not Detected
● Can Pause Trade?	No
● Pause Transfer?	No
● Max Tax?	No
● Is it Anti-whale?	No
● Is Anti-bot?	Not Detected
● Is it a Blacklist?	Yes
● Blacklist Check	Yes
● Can Mint?	No
● Is it Proxy?	No
● Can Take Ownership?	Yes
● Hidden Owner?	Not Detected
● Self Destruction?	Not Detected
● Auditor Confidence	High

Overall Audit Result: PASSED

Documentation

We were given a NBCB smart contract code in the form of an Polygonscan web link.

<https://polygonscan.com/token/0x62597B43DBEfCCb97A0c6a8efbc8A451e4828254#code>

The logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are on industry standard libraries like openzeppelin.

AS-IS overview

Functions

Sr.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	name	read	Passed	No Issue
3	symbol	read	Passed	No Issue
4	decimals	read	Passed	No Issue
5	totalSupply	read	Passed	No Issue
6	balanceOf	read	Passed	No Issue
7	transfer	write	Passed	No issue
8	allowance	Read	Passed	No Issue
9	approve	Write	Passed	No Issue
10	transferFrom	Write	Passed	No Issue
11	increaseAllowance	Write	Passed	No Issue
12	decreaseAllowance	Write	Passed	No Issue
13	_transfer	internal	Passed	No Issue
14	_mint	internal	Passed	No Issue
15	_burn	internal	Passed	No Issue
16	_approve	internal	Passed	No Issue
17	_spendAllowance	internal	Passed	No Issue
18	_beforeTokenTransfer	internal	Passed	No Issue
19	_afterTokenTransfer	internal	Passed	No Issue
20	onlyOwner	modifier	Passed	No Issue
21	owner	Read	Passed	No Issue
22	_checkOwner	internal	Passed	No Issue
23	renounceOwnership	Write	access only Owner	No Issue
24	transferOwnership	Write	access only Owner	No Issue
25	_transferOwnership	internal	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Very Low / Informational / Best practices:

No Informational severity vulnerabilities were found.

Centralization

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet private key would be compromised, then it would create trouble.

Following are Admin functions:

Ownable.sol

- `renounceOwnership`: Deleting ownership will leave the contract without an owner, removing any owner-only functionality.
- `transferOwnership`: The current owner can transfer ownership of the contract to a new account.

To make the smart contract 100% decentralized, we suggest renouncing ownership of the smart contract once its function is completed.

Conclusion

We were given a contract code in the form of Polygonscan web links. And we have used all possible tests based on given objects as files. We had no issues in the smart contract.

Since possible test cases can be unlimited for such smart contracts protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed smart contract, based on standard audit procedure scope, is **“Secured”**.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis and user interface interaction. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results and search for similar projects, examine source code dependencies.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

Infinity Blockchain Solutions.io Disclaimer

Infinity Blockchain Solutions team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: 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).

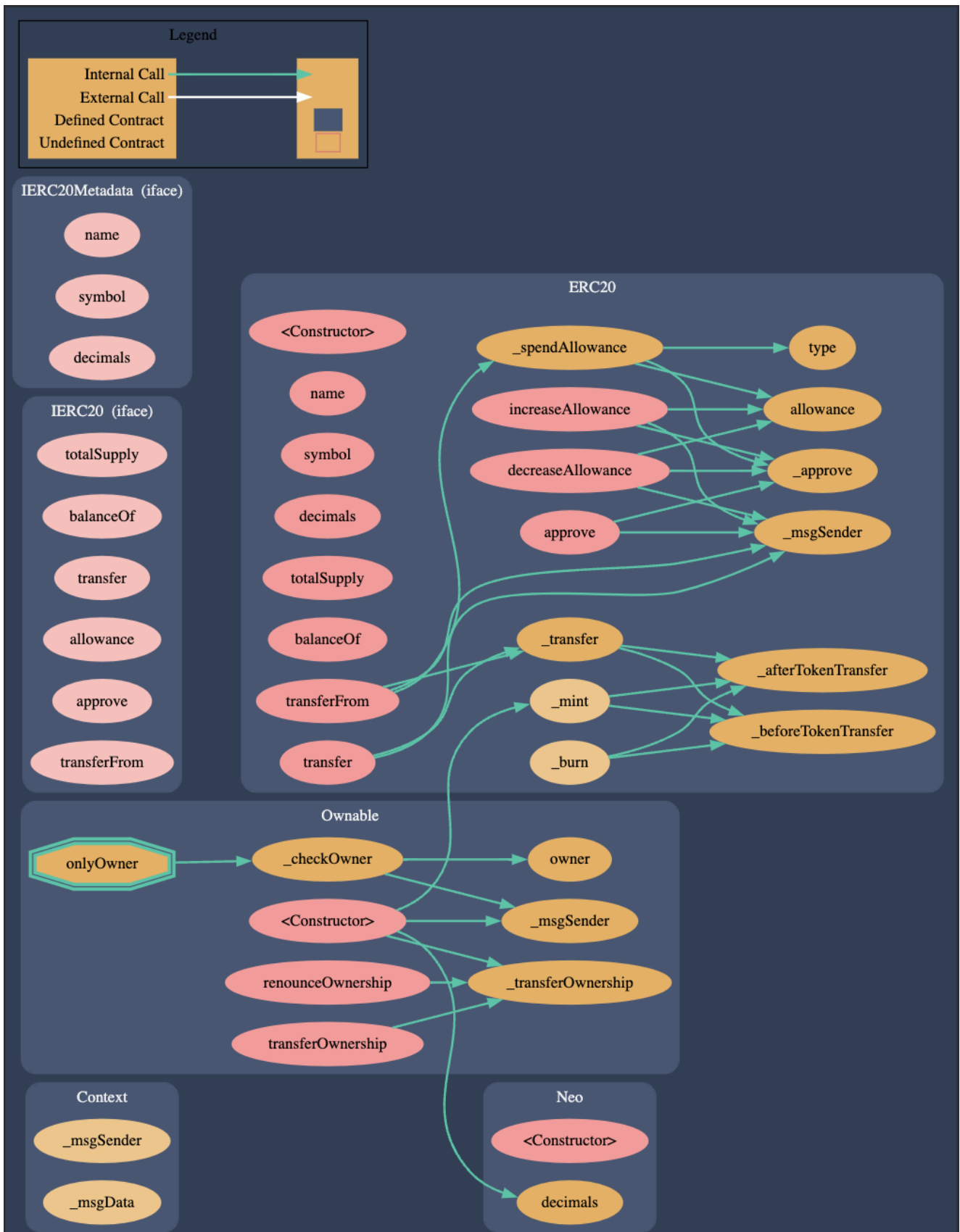
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we 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 also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Appendix

Code Flow Diagram – Neo.sol



This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of Infinity Blockchain Solutions.

Email: infinityblockchainsolutions@gmail.com

Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

Slither Log >> Neo.sol

```
Context._msgData() (contracts/Neo.sol#23-25) is never used and should be removed
ERC20._burn(address,uint256) (contracts/Neo.sol#532-548) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version0.8.22 (contracts/Neo.sol#2) necessitates a version too recent to be trusted. Consider d
eveloping with 0.6.12/0.7.6/0.8.7
solc-0.8.22 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidit
y

renounceOwnership() should be declared external:
  - Ownable.renounceOwnership() (contracts/Neo.sol#88-90)
transferOwnership(address) should be declared external:
  - Ownable.transferOwnership(address) (contracts/Neo.sol#96-102)
name() should be declared external:
  - ERC20.name() (contracts/Neo.sol#286-288)
symbol() should be declared external:
  - ERC20.symbol() (contracts/Neo.sol#294-296)
totalSupply() should be declared external:
  - ERC20.totalSupply() (contracts/Neo.sol#318-320)
balanceOf(address) should be declared external:
  - ERC20.balanceOf(address) (contracts/Neo.sol#325-329)
transfer(address,uint256) should be declared external:
  - ERC20.transfer(address,uint256) (contracts/Neo.sol#339-346)
approve(address,uint256) should be declared external:
  - ERC20.approve(address,uint256) (contracts/Neo.sol#368-375)
transferFrom(address,address,uint256) should be declared external:
  - ERC20.transferFrom(address,address,uint256) (contracts/Neo.sol#393-402)
increaseAllowance(address,uint256) should be declared external:
  - ERC20.increaseAllowance(address,uint256) (contracts/Neo.sol#416-423)
decreaseAllowance(address,uint256) should be declared external:
  - ERC20.decreaseAllowance(address,uint256) (contracts/Neo.sol#439-454)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be
-declared-external
```

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

Neo.sol

Similar variable names:

ERC20._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 509:41:

Similar variable names:

ERC20._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 509:50:

Similar variable names:

ERC20._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 511:24:

Similar variable names:

ERC20._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 514:22:

Similar variable names:

ERC20._mint(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.
Pos: 514:34:

No return:

IERC20.transfer(address,uint256): Defines a return type but never explicitly returns a value.
Pos: 158:4:

No return:

IERC20.allowance(address,address): Defines a return type but never explicitly returns a value.
Pos: 167:4:

No return:

IERC20.approve(address,uint256): Defines a return type but never explicitly returns a value.
Pos: 186:4:

No return:

IERC20.transferFrom(address,address,uint256): Defines a return type but never explicitly returns a value.
Pos: 197:4:

No return:

IERC20Metadata.name(): Defines a return type but never explicitly returns a value.
Pos: 217:4:

No return:

IERC20Metadata.symbol(): Defines a return type but never explicitly returns a value.
Pos: 222:4:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 97:8:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 445:8:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 475:8:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 476:8:

No return:

IERC20.transfer(address,uint256): Defines a return type but never explicitly returns a value.

Pos: 158:4:

No return:

IERC20.allowance(address,address): Defines a return type but never explicitly returns a value.

Pos: 167:4:

No return:

IERC20.approve(address,uint256): Defines a return type but never explicitly returns a value.

Pos: 186:4:

No return:

IERC20.transferFrom(address,address,uint256): Defines a return type but never explicitly returns a value.

Pos: 197:4:

No return:

IERC20Metadata.name(): Defines a return type but never explicitly returns a value.

Pos: 217:4:

No return:

IERC20Metadata.symbol(): Defines a return type but never explicitly returns a value.

Pos: 222:4:

Solhint Linter

Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

Neo.sol



Compiler version 0.8.22 does not satisfy the ^0.5.8 semver requirement
Pos: 1:1

Explicitly mark visibility in function (Set ignoreConstructors to true if using solidity >=0.7.0)
Pos: 5:54

Error message for require is too long
Pos: 9:96

Explicitly mark visibility in function (Set ignoreConstructors to true if using solidity >=0.7.0)
Pos: 5:277

Error message for require is too long
Pos: 9:444

Error message for require is too long
Pos: 9:474

Error message for require is too long
Pos: 9:475

Error message for require is too long
Pos: 9:480

Software analysis result:

These softwares reported many false positive results and some are informational issues. So, those issues can be safely ignored.

INFINITY

BLOCKCHAIN SOLUTIONS