# Security Audit Report for Neo X

**Date:** July 9, 2024 **Version:** 1.0
**Contact:** contact@blocksec.com

# Contents

## Report Manifest

| Item | Description |
|---|---|
| Client | Bane Labs |
| Target | Neo X |

## Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | July 9, 2024 | First release |

## Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-07-10 02:39:43+0000

**About BlockSec** BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

# Chapter 1 Introduction

## 1.1 About Target Contracts

| Information | Description |
|---|---|
| Type | Smart Contract & Modifications to Full Node Implementation |
| Language | Solidity & Go |
| Approach | Semi-automatic and manual verification |

The target of this audit is the code repository of Neo X by Bane Labs. Neo X is an EVM-compatible sidechain incorporating Neo's distinctive dBFT consensus mechanism [1]. As described in the design document [2], the dBFT protocol requires more than half (i.e., 1/2 instead of 2/3) as the consensus threshold for voting. This means that 4 validators are sufficient to achieve consensus since the top 7 candidates will be selected for each epoch.

Please note that this audit does **NOT** cover all modules in the repository. Specifically excluded are source files under the *consensus* directory, which implement the dBFT protocol. The audit scope is as follows:

- The difference between Geth 1.13.11 [3] (commit hash: `99dc3fe118a4d881d9b5347b5345669f52de8143`) and bane-go-ethereum[4] (commit hash: `971cac59f17c2c4673f8170daf2a9fc94acee74b`)
- PR #230 (latest commit hash: `e8ef782c848839bb194d6ccf0810b4181b4019f3`)
- PR #166 (latest commit hash: `9c958085ce813384d90273680a86e27044fb3ae9`)

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

| Project | Version | Commit Hash / PR |
|---|---|---|
| Neo X | `Version 1` | `971cac59f17c2c4673f8170daf2a9fc94acee74b` |
| | | PR #230 |
| | | PR #166 |
| | `Version 2` | `9af084ef711a7310f398a592ad74e8c4da207f9a` |

## 1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on,

[1] https://docs.neo.org/docs/en-us/basic/consensus/consensus_algorithm.html

[2] https://docs.banelabs.org/governance/neo-x-system-contracts

[3] https://github.com/ethereum/go-ethereum

[4] https://github.com/bane-labs/go-ethereum

the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

## 1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

### 1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

### 1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management

* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

### 1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

### 1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

**Note** *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

## 1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [5] and Common Weakness Enumeration [6]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

**Table 1.1:** Vulnerability Severity Classification

| **Impact** | | | |
|---|---|---|---|
| | *High* | High | Medium |
| | *Low* | Medium | Low |
| | | *High* | *Low* |
| | | **Likelihood** | |

[5] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[6] https://cwe.mitre.org/

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

# Chapter 2 Findings

In total, we found **three** potential security issues.

- High Risk: 1
- Medium Risk: 2

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | Medium | Potential DoS risk | Software Security | Fixed |
| 2 | High | Insufficient validation for P2P network messages | Software Security | Fixed |
| 3 | Medium | Lack of a time lock mechanism | DeFi Security | Fixed |

The details are provided in the following sections.

## 2.1 Software Security

### 2.1.1 Potential DoS risk

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `Governance` contract allows anyone to pay a registration fee to become a validator candidate. The `registerCandidate` function ensures that the number of registered candidates does not exceed a preconfigured capacity. However, the registration fee is fully refunded to the candidate upon exit via the `exitCandidate` function. This design flaw enables malicious users to perform a denial-of-service (DoS) attack on the registration process by massively registering new candidates to occupy the candidate capacity.

```
  function registerCandidate(uint shareRate) external payable {
      if (tx.origin != msg.sender) revert Errors.OnlyEOA();
      if (msg.value < registerFee) revert Errors.InsufficientValue();
      if (shareRate > 1000) revert Errors.InvalidShareRate();
      if (candidateList.length() >= IPolicy(POLICY).getCandidateLimit())
          revert Errors.RegisterDisabled();
      if (exitHeightOf[msg.sender] > 0) revert Errors.LeftNotClaimed();
      if (!candidateList.add(msg.sender)) revert Errors.CandidateExists();
      if (receivedVotes[msg.sender] > 0) {
          totalVotes += receivedVotes[msg.sender];
      }

      // record share rate and balance
      shareRateOf[msg.sender] = shareRate;
      candidateBalanceOf[msg.sender] = msg.value;
      emit Register(msg.sender);
 }

 function exitCandidate() external {
      if (!candidateList.remove(msg.sender))
          revert Errors.CandidateNotExists();
      // remove candidate list, balance still locked
      exitHeightOf[msg.sender] = block.number;
      if (receivedVotes[msg.sender] > 0) {
          totalVotes -= receivedVotes[msg.sender];
      }
      emit Exit(msg.sender);
 }
```

**Listing 2.1:** contracts/solidity/Governance.sol

**Impact** Malicious users can DoS the candidate registration process.

**Suggestion** Charge a fee to increase the cost of performing DoS attacks.

### 2.1.2 Insufficient validation for P2P network messages

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the forked version of the full node implementation, a new P2P messaging protocol called dBFT is introduced for transmitting messages related to the dBFT protocol among peers in the P2P network. Upon receipt, each message undergoes initial verification to ensure valid signatures are present. Subsequently, the message is processed by the dBFT engine and then placed into a pool for further querying by other peers. However, the function responsible for verifying whether the sender of a message is allowed, i.e., the `IsAddressAllowed` function, consistently returns true.

```
func (l *ledger) IsAddressAllowed(addr common.Address) bool {
    // Call governance contract here.
    return true
}
```

**Listing 2.2:** eth/protocols/dbft/ledger.go

**Impact** Unverified P2P messages allow malicious nodes to impersonate privileged ones, thereby increasing the resource usage of full nodes with fake messages.

**Suggestion** Verify the messages properly.

## 2.2 DeFi Security

### 2.2.1 Lack of a time lock mechanism

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `GovernanceVote` contract implements a voting mechanism that requires a privileged operation (i.e., setting key parameters of the entire network) to be voted on by a sufficient number of voters before execution. The `GovProxyAdmin` inherits this contract and adapts this mechanism to upgrade the `Governance` contract.

However, the voting mechanism lacks a time lock mechanism, which ensures that an operation can only be executed after a certain period following the receipt of enough votes. In the worst-case scenario, malicious candidates could insert themselves into the `currentConsensus` list, vote for the upgrade, and immediately profit by draining the contract. Implementing a time lock mechanism in the voting process would provide a rescue timeframe for the project if malicious proposals are passed.

```
modifier needVote(bytes32 methodKey, bytes32 paramKey) {
    address[] memory miners = IGovReward(GOV_REWARD).getMiners();
    if (!_contains(miners, msg.sender)) revert Errors.NotMiner();
    // update vote map
    _vote(methodKey, paramKey);
    // check vote, if not pass just return
    if (!_checkVote(methodKey, paramKey, miners)) {
        return;
    }
    emit VotePass(methodKey, paramKey);
    // clear vote
    _clearVote(methodKey);
    // execute method
    _;
}
```

**Listing 2.3:** contracts/solidity/base/GovernanceVote.sol

```
function upgradeAndCall(
    GovProxyUpgradeable proxy,
    address newImplementation,
    bytes memory data
)
    public
    payable
    virtual
    needVote(
        bytes32(
            // keccak256("upgradeAndCall")
            0xe739b9109d83c1c6d0d640fe9ed476fc5862a6de5483b00678a3fffa7a2be2f6
        ),
        keccak256(abi.encode(proxy, newImplementation, data))
    )
{
    proxy.upgradeToAndCall{value: msg.value}(newImplementation, data);
}
```

**Listing 2.4:** contracts/solidity/GovProxyAdmin.sol

**Impact** Malicious proposals will be executed immediately once the votes are manipulated, leaving no time window for rescues.

**Suggestion** Implement a time lock before executing the proposals.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS