SOFTWARE AUDIT REPORT

for

NEO GLOBAL DEVELOPMENT LTD.

Prepared By: Shuxiao Wang

Hangzhou, China

Jan. 4, 2021

Document Properties

Client	Neo Global Development Ltd.
Title	Software Audit Report
Target	Neo3 Blockchain
Version	0.1
Author	Edward Lo
Auditors	Edward Lo, Xudong Shao, Ruiyi Zhang
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author	Description
0.1	Jan. 4, 2021	Edward Lo	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Neo3 Blockchain** design document and related source code, we outline in this report our systematic method to evaluate potential security issues in the Neo3 Blockchain implementation, expose possible semantic inconsistencies between the source code and the design specification, and provide additional suggestions and recommendations for improvement. Our results show that the given branch of Neo3 Blockchain can be further improved due to the presence of several issues related to either security or performance. This document describes our audit results in detail.

1.1 About Neo3 Blockchain

`Neo` is an open-source, community driven platform that is leveraging the intrinsic advantages of blockchain technology to realize the optimized digital world of the future. `Neo3` is the latest version of the Neo network, a major upgrade from the earlier `Neo2` mainnet. `Neo3` features a newly designed on-chain governance and economic model, a built-in oracle, a decentralized storage system - `NeoFS`, a decentralized ID solution - `NeoID`, and an unified system architecture for optimized smart contract experience. `Neo3` will be integrated with the `Poly Network` to bring heterogeneous interoperability with other blockchains.

The basic information of Neo3 Blockchain is as follows:

Table 1.1: Basic Information of Neo3 Blockchain

Item	Description
Issuer	Neo Global Development Ltd.
Website	https://neo.org/
Type	Neo Blockchain
Platform	C#
Audit Method	White-box
Latest Audit Report	Jan. 4, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit:

- https://github.com/neo-ngd/neo-fork (aaa3dc7)
- https://github.com/neo-ngd/neo-vm-fork (bc07ff9)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/neo-project/neo (f37638b)
- https://github.com/neo-project/neo-vm (e3f1584)

1.2 About PeckShield

PeckShield Inc. [1] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products including security audits. We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

In the first phase of auditing Neo3 Blockchain, we use fuzzing to find out the corner cases that may not be covered by in-house testing. Next we do white-box auditing, in which PeckShield security auditors manually review Neo3 Blockchain design and source code, analyze them for any potential issues, and follow up with issues found in the fuzzing phase. If necessary, we design and implement individual test cases to further reproduce and verify the issues. In the following subsections, we will introduce the risk model as well as the audit procedure adopted in this report.

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [2]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, and *Low* shown in Table 1.2.

1.3.2 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by systematically finding and providing possible inputs to the target program, and then monitoring the program execution for crashes (or any unexpected results). In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing. As one of the most effective methods for exposing the presence of possible vulnerabilities, fuzzing technology has been the first choice for many security researchers in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to conduct fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the Neo3 Blockchain, we use AFL [3] as the primary tool for fuzz testing.

AFL (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Since its inception, AFL has gained growing popularity in the industry and has proved its effectiveness in discovering quite a few significant software bugs in a wide range of major software projects. The basic process of AFL fuzzing is as follows:

- Generate compile-time instrumentation to record information such as code execution path;
- Construct some input files to join the input queue, and change input files according to different strategies;

- Files that trigger a crash or timeout when executing an input file are logged for subsequent analysis;
- Loop through the above process.

Throughout the AFL testing, we will reproduce each crash based on the crash file generated by AFL. For each reported crash case, we will further analyze the root cause and check whether it is indeed a vulnerability. Once a crash case is confirmed as a vulnerability of the Neo3 Blockchain, we will further analyze it as part of the white-box audit.

1.3.3 White-box Audit

After fuzzing, we continue the white-box audit by manually analyzing source code. Here we test target software's internal structure, design, coding, and we focus on verifying the flow of input and output through the application as well as examining possible design and implementation trade-offs for strengthened security. PeckShield auditors first fully review and understand the source code, then create specific test cases, execute them and analyze the results. Issues such as internal security loopholes, unexpected output, broken or poorly structured paths, etc., will be inspected under close scrutiny.

Blockchain is a secure method of creating a distributed database of transactions, and three major technologies of blockchain are cryptography, decentralization, and consensus model. Blockchain does come with unique security challenges, and based on our understanding of blockchain general design, we in this audit divide the blockchain software into the following major areas and inspect each area accordingly:

- Data and state storage, which is related to the database and files where blockchain data are saved.
- P2P networking, consensus, and transaction model in the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.
- VM, account model, and incentive model. This is essentially the execution and business layer of the blockchain, and many blockchain business specific logics are implemented here.
- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.
- Others. This includes any software modules that do not belong to above-mentioned areas, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Table 1.3: The Full List of Audited Items (Part I)

Category	Check Item
Data and State Storage	Blockchain Database Security
	Database State Integrity Check
Node Operation	Default Configuration Security
	Default Configuration Optimization
	Node Upgrade And Rollback Mechanism
Node Communication	External RPC Implementation Logic
	External RPC Function Security
	Node P2P Protocol Implementation Logic
	Node P2P Protocol Security
	Serialization/Deserialization
	Invalid/Malicious Node Management Mechanism
	Communication Encryption/Decryption
	Eclipse Attack Protection
	Fingerprint Attack Protection
Consensus	Consensus Algorithm Scalability
	Consensus Algorithm Implementation Logic
	Consensus Algorithm Security
Transaction Model	Transaction Privacy Security
	Transaction Fee Mechanism Security
	Transaction Congestion Attack Protection
VM	VM Implementation Logic
	VM Implementation Security
	VM Sandbox Escape
	VM Stack/Heap Overflow
	Contract Privilege Control
	Predefined Function Security
Account Model	Status Storage Algorithm Adjustability
	Status Storage Algorithm Security
	Double Spending Protection
Incentive Model	Mining Algorithm Security
	Mining Algorithm ASIC Resistance
	Tokenization Reward Mechanism

Table 1.4: The Full List of Audited Items (Part II)

Category	Check Item
System Contracts And Services	Memory Leak Detection
	Use-After-Free
	Null Pointer Dereference
	Undefined Behaviors
	Deprecated API Usage
	Signature Algorithm Security
	Multisignature Algorithm Security
SDK Security	Using RPC Functions Security
	Privatekey Algorithm Security
	Communication Security
	Function integrity checking code
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

Based on the above classification, we show in Table 1.3 and Table 1.4 the detailed list of the audited items in this report.

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [4], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.5 to classify our findings.

Table 1.5: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Input Validation Issues	Weaknesses in this category are related to a software system's input validation components. Frequently these deal with sanitizing, neutralizing and validating any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given blockchain software, 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 several independent audits and a public bug bounty program to ensure the security of blockchain software. Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Neo3 Blockchain implementation. During the first phase of our audit, we study the source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tools. After that, we manually review business logics, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	4	■■■■
High	2	■■
Medium	3	■■■
Low	0	
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, the Neo3 Blockchain are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 4 critical-severity vulnerabilities, 2 high-severity vulnerabilities, and 3 medium-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Missed Limit On The Maximum Size of TryStack	Business Logic Errors	Fixed
PVE-002	Medium	Incorrect GAS Calculation In System.Contract.Update Syscall	Business Logic Errors	Fixed
PVE-003	Critical	Lack of LastBlockIndex Check in the Ping / Pong Payload	Business Logic Errors	Confirmed
PVE-004	High	Out-Of-Memory Vulnerability In The Process of GetBlockByIndex Request	Business Logic Errors	Confirmed
PVE-005	High	Out-Of-Memory Vulnerability In OnNewBlock()	Business Logic Errors	Fixed
PVE-006	Critical	Inappropriate Hash Calculation of RecoveryRequest	Business Logic Errors	Confirmed
PVE-007	Critical	No Penalty On the Misbehaving Committee Members	Business Logic Errors	Confirmed
PVE-008	Medium	Inappropriate Hash Calculation of Transactions	Coding Practices	Confirmed
PVE-009	Medium	Potentially Halted Consensus By The Malicious Speaker	Coding Practices	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Chapter 3 for details.

3 Detailed Results

3.1 Missed Limit On The Maximum Size of TryStack

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: neo-vm/ExecutionEngine.cs
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-770 [6]

Description

NeoVM is designed for executing smart contracts. It uses a normal reference counting model to limit the number of stackItem used in it. In current implementation, the reference count cannot exceed MaxStackSize(2 * 1024).

```
    protected virtual void PostExecuteInstruction()
    {
        if (ReferenceCounter.CheckZeroReferred() > MaxStackSize)
            throw new InvalidOperationException($"MaxStackSize exceed: {ReferenceCounter
                .Count}");
    }
```

Listing 3.1: neo-vm/ExecutionEngine.cs

```
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Push(StackItem item)
    {
        innerList.Add(item);
        referenceCounter.AddStackReference(item);
    }
```

Listing 3.2: neo-vm/EvaluationStack.cs

Meanwhile, the referenceCounter is shared between contexts. Specifically, when the engine executes the instruction —— OpCode.CALL, a new executionContext will be cloned with the current referenceCounter and pushed into the InvocationStack. With such a mechanism, the memory usage of NeoVM will be limited to a proper range. In the following, we show the related code snippet:

```
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void ExecuteCall(int position)
    {
        LoadContext(CurrentContext.Clone(position));
    }
```

Listing 3.3: neo-vm/ExecutionEngine.cs

```
    public ExecutionContext Clone()
    {
        return Clone(InstructionPointer);
    }

    public ExecutionContext Clone(int initialPosition)
    {
        return new ExecutionContext(shared_states, initialPosition);
    }
```

Listing 3.4: neo-vm/ExecutionContext.cs

```
    public SharedStates(Script script, ReferenceCounter referenceCounter)
    {
        this.Script = script;
        this.EvaluationStack = new EvaluationStack(referenceCounter);
        this.States = new Dictionary<Type, object>();
    }
```

Listing 3.5: neo-vm/ExecutionContext.SharedStates.cs

However, when the engine executes the instruction —— `OpCode.TRY`, a new `ExceptionHandlingContext` will be created without adding the reference counts. Specifically, an attacker can craft the `tx.script` that makes the VM keep executing the `OpCode.TRY` in a loop. Now, the `MaxBlockSystemFee` is `9000 * (long)GAS.Factor`. If the victim node receives the malicious tx with sufficient gas, it will consume lots of memories and trigger `Out-Of-Memory` eventually.

```
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void ExecuteTry(int catchOffset, int finallyOffset)
    {
        if (catchOffset == 0 && finallyOffset == 0)
            throw new InvalidOperationException($"catchOffset and finallyOffset can't be
                0 in a TRY block");
        int catchPointer = catchOffset == 0 ? -1 : checked(CurrentContext.
            InstructionPointer + catchOffset);
        int finallyPointer = finallyOffset == 0 ? -1 : checked(CurrentContext.
            InstructionPointer + finallyOffset);
        CurrentContext.TryStack ??= new Stack<ExceptionHandlingContext>();
        CurrentContext.TryStack.Push(new ExceptionHandlingContext(catchPointer,
            finallyPointer));
    }
```

Listing 3.6: neo-vm/ExecutionEngine.cs

Recommendation Apply the `MaximumTryStackSize` threshold to avoid `Out-Of-Memory`.

Status The issue has been fixed by this commit: `e3f1584`.

3.2 Incorrect GAS Calculation In System.Contract.Update Syscall

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: low
- Target: `ApplicationEngine.Contract.cs`
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-841 [7]

Description

As mentioned in Section 3.1, NeoVM is designed for executing smart contracts and has a built-in gas cost model for each supported instruction. The price of `SYSCALL` is set to 0 and defined individually in the implementation of each syscall. In this section, we examine a vulnerability in `System.Contract.Update` syscall that could lead to an incorrect GAS calculation.

```
internal void UpdateContract(byte[] script, byte[] manifest)
{
    AddGas(StoragePrice * (script?.Length ?? 0 + manifest?.Length ?? 0));

    var contract = Snapshot.Contracts.TryGet(CurrentScriptHash);
    if (contract is null) throw new InvalidOperationException($"Updating Contract Does Not Exist: {CurrentScriptHash}");

    if (script != null)
    {
        if (script.Length == 0 || script.Length > MaxContractLength)
            throw new ArgumentException($"Invalid Script Length: {script.Length}");
        UInt160 hash_new = script.ToScriptHash();
        if (hash_new.Equals(CurrentScriptHash) || Snapshot.Contracts.TryGet(hash_new) != null)
            throw new InvalidOperationException($"Adding Contract Hash Already Exist: {hash_new}");
        contract = new ContractState
        {
            Id = contract.Id,
            Script = script.ToArray(),
            Manifest = contract.Manifest
        };
        contract.Manifest.Abi.Hash = hash_new;
        Snapshot.Contracts.Add(hash_new, contract);
        Snapshot.Contracts.Delete(CurrentScriptHash);
```

```
    }
    if (manifest != null)
    {
        if (manifest.Length == 0 || manifest.Length > ContractManifest.MaxLength)
            throw new ArgumentException($"Invalid Manifest Length: {manifest.Length}");
        contract = Snapshot.Contracts.GetAndChange(contract.ScriptHash);
        contract.Manifest = ContractManifest.Parse(manifest);
        if (!contract.Manifest.IsValid(contract.ScriptHash))
            throw new InvalidOperationException($"Invalid Manifest Hash: {contract.
                ScriptHash}");
        if (!contract.HasStorage && Snapshot.Storages.Find(BitConverter.GetBytes(
            contract.Id)).Any())
            throw new InvalidOperationException($"Contract Does Not Support Storage But
                Uses Storage");
    }
}
```

Listing 3.7: neo/SmartContract/ApplicationEngine.Contract.cs

Specifically, the function `UpdateContract()` does not validate whether the given `script` and `manifest` are null. If they are null, the final price will be computed as 0. In other words, the related `System.Contract.Update` syscall will not cost any gas.

Recommendation Add a sanity check on the given `script` and `manifest` parameters in `UpdateContract()`.

Status The issue has been fixed by this commit: `127272f`.

3.3 Lack of LastBlockIndex Check in the Ping / Pong Payload

- ID: PVE-003
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Network/P2P/RemoteNode.ProtocolHandler.cs`
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-770 [6]

Description

In the underlying P2P network of `Neo`, various information is packed as `InvPayload` for transmission and exchange. Different payloads have their own specific formats and contents. In current implementation, there are three types of inventories as the payloads:

- Transaction (`InventoryType` = $0x2b$)
- Block (`InventoryType` = $0x2c$)

- Consensus (`InventoryType` $= 0x2d$)

```
297 private void OnInvMessageReceived(InvPayload payload)
298 {
299     UInt256[] hashes = payload.Hashes.Where(p => !pendingKnownHashes.Contains(p) && !
            knownHashes.Contains(p) && !sentHashes.Contains(p)).ToArray();
300     if (hashes.Length == 0) return;
301     switch (payload.Type)
302     {
303         case InventoryType.Block:
304             using (SnapshotView snapshot = Blockchain.Singleton.GetSnapshot())
305                 hashes = hashes.Where(p => !snapshot.ContainsBlock(p)).ToArray();
306             break;
307         case InventoryType.TX:
308             using (SnapshotView snapshot = Blockchain.Singleton.GetSnapshot())
309                 hashes = hashes.Where(p => !snapshot.ContainsTransaction(p)).ToArray();
310             break;
311     }
312     if (hashes.Length == 0) return;
313     foreach (UInt256 hash in hashes)
314         pendingKnownHashes.Add((hash, DateTime.UtcNow));
315     system.TaskManager.Tell(new TaskManager.NewTasks { Payload = InvPayload.Create(
            payload.Type, hashes) });
316 }
```

Listing 3.8: Network/P2P/RemoteNode.ProtocolHandler.cs

To elaborate, we show above the `OnInvMessageReceived()` routine that, as the name indicates, is tasked with handling received messages. Specifically, this `OnInvMessageReceived()` routine will be called when a `MessageCommand.Inv` message is received. Apparently, it will filter out hashes that are already known or sent (lines 299-312), and the remaining hashes will be packaged in a `NewTasks` message and sent to TaskManager. (line 315)

```
private void OnNewTasks(InvPayload payload)
{
    if (!sessions.TryGetValue(Sender, out TaskSession session))
        return;
    // Do not accept payload of type InventoryType.TX if not synced on best known
        HeaderHeight
    if (payload.Type == InventoryType.TX && Blockchain.Singleton.Height < sessions.
        Values.Max(p => p.LastBlockIndex))
        return;
    HashSet<UInt256> hashes = new HashSet<UInt256>(payload.Hashes);
    // Remove all previously processed knownHashes from the list that is being requested
    hashes.Remove(knownHashes);

    // Remove those that are already in process by other sessions
    hashes.Remove(globalTasks);
    if (hashes.Count == 0)
        return;

```

```
107     // Update globalTasks with the ones that will be requested within this current
            session
108     foreach (UInt256 hash in hashes)
109     {
110         IncrementGlobalTask(hash);
111         session.InvTasks[hash] = DateTime.UtcNow;
112     }
113
114     foreach (InvPayload group in InvPayload.CreateGroup(payload.Type, hashes.ToArray()))
115         Sender.Tell(Message.Create(MessageCommand.GetData, group));
116 }
```

Listing 3.9: Network/P2P/RemoteNode.ProtocolHandler.cs

Each new incoming message is encapsulated and scheduled for processing in a a new task via the `OnNewTasks()` routine. This routine has a rather straightforward business logic in performing some sanity checks, filtering out already known hashes, and querying remote node for further data (line 115). However, the flow may be compromised by a malicious attacker. Specifically, if the message type has the payload type of `InventoryType.TX`, the victim node will not accept the message if it has not synchronized with the best known `HeaderHeight` (line 96). The best known `HeaderHeight` is the max `LastBlockIndex` among the connected sessions and is determined from the received `VersionPayload` while establishing the connection, or can be modified later using `Ping/Pong` message. So an attacker can intentionally provide a large `StartHeight` in the `VersionPayload` or using the `Ping/Pong` message to manipulate it after establishing the connection. As a result, the victim node will not handle `InventoryType.TX` messages anymore.

Recommendation Add a sanity check on `LastBlockIndex` in the `Ping/Pong` message or `StartHeight` in the `VersionPayload`.

Status The issue has been confirmed.

3.4 Out-Of-Memory Vulnerability In The Process of GetBlockByIndex Request

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Network/P2P/RemoteNode.ProtocolHandler.cs`
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-770 [6]

Description

There are many different types of payloads defined in the `Neo` P2P network and these payloads are transferred for information exchange between nodes in the P2P network. Meanwhile, `Neo` also defines other types to synchronize communicating nodes. For example, the `GetBlockByIndexPayload` message type is used for a node to ask for blocks from its neighbors.

```
private void OnGetBlockByIndexMessageReceived(GetBlockByIndexPayload payload)
{
    uint count = payload.Count == -1 ? InvPayload.MaxHashesCount : Math.Min((uint)
        payload.Count, InvPayload.MaxHashesCount);
    for (uint i = payload.IndexStart, max = payload.IndexStart + count; i < max; i++)
    {
        Block block = Blockchain.Singleton.GetBlock(i);
        if (block == null)
            break;

        if (bloom_filter == null)
        {
            EnqueueMessage(Message.Create(MessageCommand.Block, block));
        }
        else
        {
            BitArray flags = new BitArray(block.Transactions.Select(p => bloom_filter.
                Test(p)).ToArray());
            EnqueueMessage(Message.Create(MessageCommand.MerkleBlock, MerkleBlockPayload
                .Create(block, flags)));
        }
    }
}
```

Listing 3.10: Network/P2P/RemoteNode.ProtocolHandler.cs

This message is handled by the `GetBlockByIndexMessageReceived()` routine, which will be triggered when a `MessageCommand.GetBlockByIndex` message is received. This routine will return the specified number of blocks starting with the requested `IndexStart` to the `RemoteNode` actor. A limit set by

HeadersPayload.MaxHashesCount (500) is also applied to the number of requested Headers, namely payload.Count.

```
68  private void EnqueueMessage(Message message)
69  {
70      bool is_single = false;
71      switch (message.Command)
72      {
73          case MessageCommand.Addr:
74          case MessageCommand.GetAddr:
75          case MessageCommand.GetBlocks:
76          case MessageCommand.GetHeaders:
77          case MessageCommand.Mempool:
78          case MessageCommand.Ping:
79          case MessageCommand.Pong:
80              is_single = true;
81              break;
82      }
83      Queue<Message> message_queue;
84      switch (message.Command)
85      {
86          case MessageCommand.Alert:
87          case MessageCommand.Consensus:
88          case MessageCommand.FilterAdd:
89          case MessageCommand.FilterClear:
90          case MessageCommand.FilterLoad:
91          case MessageCommand.GetAddr:
92          case MessageCommand.Mempool:
93              message_queue = message_queue_high;
94              break;
95          default:
96              message_queue = message_queue_low;
97              break;
98      }
99      if (!is_single || message_queue.All(p => p.Command != message.Command))
100         message_queue.Enqueue(message);
101     CheckMessageQueue();
102 }
```

Listing 3.11: Network/P2P/RemoteNode.cs

The message will be filtered based on whether it is a one-at-a-time message (lines 71-82), and put into the corresponding queue according to its priority (lines 84-98). However, there is no limitation on the MessageCommand.GetBlockByIndex request a node can send, nor has it on the MessageCommand.Block or MessageCommand.MerkleBlock.

Therefore, if an attacker keeps sending GetBlockByIndex requests to a chosen target, the target node will try to enqueue the corresponding Block or MerkleBlock into message_queue_low, which could consumes lots of memories, and eventually cause the victim nodes Out-Of-Memory.

Recommendation Add a size limitation on message_queue_low.

Status The issue has been confirmed.

3.5 Out-Of-Memory Vulnerability In OnNewBlock()

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Ledger/Blockchain.cs`
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-770 [6]

Description

As we stated in Section 3.3, the `InvPayload` is used for information exchange between nodes, and the contents of the corresponding inventory, i.e., block / transaction / consensus, are packed in the `Block`, `Transaction` and `ConsensusPayload` and transferred between each node and the neighbor peers.

```
private void OnMessage(Message msg)
 {
     foreach (IP2PPlugin plugin in Plugin.P2PPlugins)
         if (!plugin.OnP2PMessage(msg))
             return;
     if (Version == null)
     {
         if (msg.Command != MessageCommand.Version)
             throw new ProtocolViolationException();
         OnVersionMessageReceived((VersionPayload)msg.Payload);
         return;
     }
     if (!verack)
     {
         if (msg.Command != MessageCommand.Verack)
             throw new ProtocolViolationException();
         OnVerackMessageReceived();
         return;
     }
     switch (msg.Command)
     {
         case MessageCommand.Addr:
             OnAddrMessageReceived((AddrPayload)msg.Payload);
             break;
         case MessageCommand.Block:
             OnInventoryReceived((Block)msg.Payload);
             break;
```

Listing 3.12: Network/P2P/RemoteNode.ProtocolHandler.cs

To elaborate, we show the `OnMessage()` routine that handles the incoming messages. If the message has the command `MessageCommand.Block`, the routine calls a helper function, i.e., `OnInventoryReceived()`.

```
private void OnInventoryReceived(IInventory inventory)
{
    system.TaskManager.Tell(inventory);
    if (inventory is Transaction transaction)
        system.Consensus?.Tell(transaction);
    system.Blockchain.Tell(inventory, ActorRefs.NoSender);
    pendingKnownHashes.Remove(inventory.Hash);
    knownHashes.Add(inventory.Hash);
}
```

Listing 3.13: Network/P2P/RemoteNode.ProtocolHandler.cs

After that, two additional functions`OnInventory()` and `OnNewBlock()` are called to add the block.

```
private void OnInventory(IInventory inventory, bool relay = true)
{
    RelayResult rr = new RelayResult
    {
        Inventory = inventory,
        Result = inventory switch
        {
            Block block => OnNewBlock(block),
            Transaction transaction => OnNewTransaction(transaction),
            _ => OnNewInventory(inventory)
        }
    };
    if (relay && rr.Result == VerifyResult.Succeed)
        system.LocalNode.Tell(new LocalNode.RelayDirectly { Inventory = inventory });
    Sender.Tell(rr);
    Context.System.EventStream.Publish(rr);
}

private VerifyResult OnNewBlock(Block block)
{
    if (block.Index <= Height)
        return VerifyResult.AlreadyExists;
    if (block.Index - 1 > Height)
    {
        AddUnverifiedBlockToCache(block);
        return VerifyResult.UnableToVerify;
    }
    if (block.Index == Height + 1)
    {
        if (!block.Verify(currentSnapshot))
            return VerifyResult.Invalid;
        block_cache.TryAdd(block.Hash, block);
        block_cache_unverified.Remove(block.Index);
        // We can store the new block in block_cache and tell the new height to other
            nodes before Persist().
```

```
        system.LocalNode.Tell(Message.Create(MessageCommand.Ping, PingPayload.Create(Singleton.Height + 1)));
        Persist(block);
        SaveHeaderHashList();
        if (block_cache_unverified.TryGetValue(Height + 1, out LinkedList<Block> unverifiedBlocks))
        {
            foreach (var unverifiedBlock in unverifiedBlocks)
                Self.Tell(unverifiedBlock, ActorRefs.NoSender);
            block_cache_unverified.Remove(Height + 1);
        }
    }
    return VerifyResult.Succeed;
}
```

Listing 3.14: Ledger/Blockchain.cs

If the block has its index bigger than the current block height plus 1, the block will be added to an unverified cache directly via `AddUnverifiedBlockToCache`.

```
private void AddUnverifiedBlockToCache(Block block)
{
    // Check if any block proposal for height `block.Index` exists
    if (!block_cache_unverified.TryGetValue(block.Index, out LinkedList<Block> blocks))
    {
        // There are no blocks, a new LinkedList is created and, consequently, the current block is added to the list
        blocks = new LinkedList<Block>();
        block_cache_unverified.Add(block.Index, blocks);
    }
    // Check if any block with the hash being added already exists on possible candidates to be processed
    foreach (var unverifiedBlock in blocks)
    {
        if (block.Hash == unverifiedBlock.Hash)
            return;
    }
    blocks.AddLast(block);
}
```

Listing 3.15: Ledger/Blockchain.cs

However, the `block_cache_unverified` is a dictionary and there is no size limitation on it. Therefore, an attacker can keep sending blocks with a large block index to flood the `block_cache_unverified`, hence causing the node `Out-Of-Memory`. Furthermore, since the `block_cache_unverified` stores blocks with same index but different hashes into a `linked list`, an attacker can also flood the node with blocks that has the same index but different hashes.

Recommendation Add a limitation on `block_cache_unverified`.

Status The issue has been fixed by this commit: `5bea103`.

3.6 Inappropriate Hash Calculation of RecoveryRequest

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Consensus/ConsensusService.cs`
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-770 [6]

Description

The `Neo` blockchain proposes the `dBFT (delegated Byzantine Fault Tolerance)` consensus algorithm based on `PBFT (Practical Byzantine Fault Tolerance)` algorithm. The algorithm `dBFT` determines the validator set according to real-time blockchain voting, which effectively enhances the effectiveness of the algorithm and has the benefits of bringing block time and transaction confirmation time savings. The `dBFT` consensus procedure can be summarized as the following steps:

1. `Speaker` starts consensus by broadcasting a `Prepare Request` message.
2. `Delegates` broadcast `Prepare Response` after receiving the `Prepare Request` message.
3. `Validators` broadcast `Commit` after receiving enough `Prepare Response` messages.
4. `Validators` produce and broadcast a new block after receiving enough `Commit` messages.

```
private void OnConsensusPayload(ConsensusPayload payload)
{
    if (context.BlockSent) return;
    if (payload.Version != context.Block.Version) return;
    if (payload.PrevHash != context.Block.PrevHash || payload.BlockIndex != context.
        Block.Index)
    {
        if (context.Block.Index < payload.BlockIndex)
        {
            Log($"chain sync: expected={payload.BlockIndex} current={context.Block.Index
                 - 1} nodes={LocalNode.Singleton.ConnectedCount}", LogLevel.Warning)
                ;
        }
        return;
    }
    if (payload.ValidatorIndex >= context.Validators.Length) return;
    ConsensusMessage message;
    try
    {
        message = payload.ConsensusMessage;
    }
    catch (FormatException)
    {
```

```
            return;
        }
        catch (IOException)
        {
            return;
        }
        context.LastSeenMessage[payload.ValidatorIndex] = (int)payload.BlockIndex;
        foreach (IP2PPlugin plugin in Plugin.P2PPlugins)
            if (!plugin.OnConsensusMessage(payload))
                return;
        switch (message)
        {
            case ChangeView view:
                OnChangeViewReceived(payload, view);
                break;
            case PrepareRequest request:
                OnPrepareRequestReceived(payload, request);
                break;
            case PrepareResponse response:
                OnPrepareResponseReceived(payload, response);
                break;
            case Commit commit:
                OnCommitReceived(payload, commit);
                break;
            case RecoveryRequest _:
                OnRecoveryRequestReceived(payload);
                break;
            case RecoveryMessage recovery:
                OnRecoveryMessageReceived(payload, recovery);
                break;
        }
    }
```

Listing 3.16: Consensus/ConsensusService.cs

When the `RecoveryRequest` message is received, the handler routine, i.e., `OnConsensusPayload()`, will firstly perform necessary basic checks, and then pass the message to `OnRecoveryRequestReceived()`.

```
private void OnRecoveryRequestReceived(ConsensusPayload payload)
{
    // We keep track of the payload hashes received in this block, and don't respond with recovery
    // in response to the same payload that we already responded to previously.
    // ChangeView messages include a Timestamp when the change view is sent, thus if a node restarts
    // and issues a change view for the same view, it will have a different hash and will correctly respond
    // again; however replay attacks of the ChangeView message from arbitrary nodes will not trigger an
    // additional recovery message response.
    if (!knownHashes.Add(payload.Hash)) return;

    Log($"On{payload.ConsensusMessage.GetType().Name}Received: height={payload.
```

```
        BlockIndex} index={payload.ValidatorIndex} view={payload.ConsensusMessage.
        ViewNumber}");
396     if (context.WatchOnly) return;
397     if (!context.CommitSent)
398     {
399         bool shouldSendRecovery = false;
400         int allowedRecoveryNodeCount = context.F;
401         // Limit recoveries to be sent from an upper limit of 'f' nodes
402         for (int i = 1; i <= allowedRecoveryNodeCount; i++)
403         {
404             var chosenIndex = (payload.ValidatorIndex + i) % context.Validators.Length;
405             if (chosenIndex != context.MyIndex) continue;
406             shouldSendRecovery = true;
407             break;
408         }
409 
410         if (!shouldSendRecovery) return;
411     }
412     Log($"send recovery: view={context.ViewNumber}");
413     localNode.Tell(new LocalNode.SendDirectly { Inventory = context.MakeRecoveryMessage
            () });
414 }
```

Listing 3.17: Consensus/ConsensusService.cs

The `OnRecoveryRequestReceived()` routine further validates whether the request has been handled or not (line 393), which is determined based on the message's calculated hash. However, the recovery request message contains a timestamp field, so the attacker can easily bypass the hash check by modifying this field, and let the victim consensus node send out recovery message to perform a network reflection attack.

Recommendation Include the `timestamp` field when calculating the payload hash.

Status The issue has been confirmed.

3.7 No Penalty On the Misbehaving Committee Members

- ID: PVE-007
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Consensus/ConsensusService.cs`
- Category: Business Logic Errors [5]
- CWE subcategory: CWE-770 [6]

Description

As we described in Section 3.6, committee members exchange information in different consensus phases (`Commit`, `Change View`, etc.). While receiving consensus payloads, the node will pass it to

`OnInventory()` of the `Blockchain` actor.

```
303 private void OnInventory(IInventory inventory, bool relay = true)
304 {
305     RelayResult rr = new RelayResult
306     {
307         Inventory = inventory,
308         Result = inventory switch
309         {
310             Block block => OnNewBlock(block),
311             Transaction transaction => OnNewTransaction(transaction),
312             _ => OnNewInventory(inventory)
313         }
314     };
315     if (relay && rr.Result == VerifyResult.Succeed)
316         system.LocalNode.Tell(new LocalNode.RelayDirectly { Inventory = inventory });
317     Sender.Tell(rr);
318     Context.System.EventStream.Publish(rr);
319 }
```

Listing 3.18: Ledger/Blockchain.cs

```
350 private VerifyResult OnNewInventory(IInventory inventory)
351 {
352     if (!inventory.Verify(currentSnapshot)) return VerifyResult.Invalid;
353     RelayCache.Add(inventory);
354     return VerifyResult.Succeed;
355 }
```

Listing 3.19: Ledger/Blockchain.cs

The consensus-related payload will be handled by the `OnNewInventory()` routine, and if it passes `Verify()` (line 352), the payload will be added to `RelayCache` (line 353) and relayed to other nodes (line 316).

```
121 public bool Verify(StoreView snapshot)
122 {
123     if (BlockIndex <= snapshot.Height)
124         return false;
125     return this.VerifyWitnesses(snapshot, 0_02000000);
126 }
```

Listing 3.20: Network/P2P/Payloads/ConsensusPayload.cs

The verification is performed by firstly checking whether the payload is not an old one (line 1234), and then returning the result of `VerifyWitnesses`. For the consensus payload, `VerifyWitnesses` will also verify the validity of the signature.

However, there is no penalty on the misbehaving committee members. Specifically, a committee member can sign any consensus payload with a higher `BlockIndex` and send it to other nodes. As long as the signature is correct, the receiving nodes will relay it to other full nodes, whether the contents are legit or not, and the same flow goes on and on. This is a typical reflection attack. In the end, this reflection attack could stuck the entire `Neo` network with bogus payloads and significantly compromise the performance of the consensus.

Recommendation Penalize misbehaving committee members.

Status The issue has been confirmed.

3.8 Inappropriate hash calculation of transactions

- ID: PVE-008
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Network/P2P/Payloads/Transaction.cs`
- Category: Coding Practices [8]
- CWE subcategory: CWE-20 [9]

Description

As mentioned in Section 3.6, the `Neo` blockchain proposes `dBFT` consensus algorithm based on the `PBFT` algorithm. When receiving a `Prepare Request` message from `Speaker`, a validator attempts to acquire corresponding transactions from its memory pool or unverified transaction pool for each `TransactionHashes` within the `request` message, and add these transactions to its consensus context. If there are any missed transactions, the validator will send the `GetData` message to ask them from other nodes. Afterwards, the validator performs sanity checks on these transactions (lines 67-82) and adds those transactions to the context if passed (lines 83-84). Conversely, if the verification failed, a `change view` request will be sent by the validator (lines 70-81). For illustration, we show below the related code snippet:

```
56     private bool AddTransaction(Transaction tx, bool verify)
57     {
58         if (verify)
59         {
60             VerifyResult result = tx.Verify(context.Snapshot, context.
                   VerificationContext);
61             if (result == VerifyResult.PolicyFail)
62             {
63                 Log($"reject tx: {tx.Hash}{Environment.NewLine}{tx.ToArray().ToHexString
                       ()}", LogLevel.Warning);
64                 RequestChangeView(ChangeViewReason.TxRejectedByPolicy);
65                 return false;
66             }
67             else if (result != VerifyResult.Succeed)
68             {
69                 Log($"Invalid transaction: {tx.Hash}{Environment.NewLine}{tx.ToArray().
                       ToHexString()}", LogLevel.Warning);
```

```
                RequestChangeView(ChangeViewReason.TxInvalid);
                return false;
            }
        }
        context.Transactions[tx.Hash] = tx;
        context.VerificationContext.AddTransaction(tx);
        return CheckPrepareResponse();
    }
```

Listing 3.21: Consensus/ConsensusService.cs

```
    public virtual VerifyResult VerifyForEachBlock(StoreView snapshot, TransactionVerificationContext context)
    {
        if (ValidUntilBlock <= snapshot.Height || ValidUntilBlock > snapshot.Height + MaxValidUntilBlockIncrement)
            return VerifyResult.Expired;
        UInt160[] hashes = GetScriptHashesForVerifying(snapshot);
        if (NativeContract.Policy.GetBlockedAccounts(snapshot).Intersect(hashes).Any())
            return VerifyResult.PolicyFail;
        if (NativeContract.Policy.GetMaxBlockSystemFee(snapshot) < SystemFee)
            return VerifyResult.PolicyFail;
        if (!(context?.CheckTransaction(this, snapshot) ?? true)) return VerifyResult.InsufficientFunds;
        if (hashes.Length != Witnesses.Length) return VerifyResult.Invalid;
        for (int i = 0; i < hashes.Length; i++)
        {
            if (Witnesses[i].VerificationScript.Length > 0) continue;
            if (snapshot.Contracts.TryGet(hashes[i]) is null) return VerifyResult.Invalid;
        }
        return VerifyResult.Succeed;
    }

    public virtual VerifyResult Verify(StoreView snapshot, TransactionVerificationContext context)
    {
        VerifyResult result = VerifyForEachBlock(snapshot, context);
        if (result != VerifyResult.Succeed) return result;
        int size = Size;
        if (size > MaxTransactionSize) return VerifyResult.Invalid;
        long net_fee = NetworkFee - size * NativeContract.Policy.GetFeePerByte(snapshot);
        if (net_fee < 0) return VerifyResult.InsufficientFunds;
        if (!this.VerifyWitnesses(snapshot, net_fee)) return VerifyResult.Invalid;
        return VerifyResult.Succeed;
    }
```

Listing 3.22: Network/P2P/Payloads/Transaction.cs

As shown in the above code, the verification process for transaction is performed by the `Verify` and `VerifyForEachBlock`. These routines will ensure the transaction follows the consensus policy, and the sender has enough funds. However, there is a lack of sanity check for the transaction hash integrity, which could be exploited by attackers to craft invalid transactions. Specifically, as shown in the code snippet below, a transaction's hash is determined by a number of fields, and a malicious attacker can modify its witnesses without compromising the hash value, and it will still be seemed as the same transaction in `TransactionHashes`. However, the malformed transaction will fail the verification, so validators have to send `change view` request and the consensus process is thus be compromised.

```
    void IVerifiable.SerializeUnsigned(BinaryWriter writer)
    {
        writer.Write(Version);
        writer.Write(Nonce);
        writer.Write(SystemFee);
        writer.Write(NetworkFee);
        writer.Write(ValidUntilBlock);
        writer.Write(Signers);
        writer.Write(Attributes);
        writer.WriteVarBytes(Script);
    }
```

Listing 3.23: Network/P2P/Payloads/Transaction.cs

Recommendation Include `Transaction.Witness` when calculating the hash.

Status This issue has been confirmed.

3.9 Potentially Halted Consensus By The Malicious Speaker

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `neo/Consensus/ConsensusService.cs`
- Category: Coding Practices [8]
- CWE subcategory: CWE-20 [9]

Description

As mentioned in in Section 3.6,the `Neo` blockchain proposes `dBFT` consensus algorithm based on the `PBFT` algorithm. It comes to our attention that `delegates` will send out the `commit` message and move into the `commit` phase after they receive `2/3 validators' Prepare Responses`. Conversely, if the `delegate` is in the `commit` phase and has already sent out the `commit` message, it will not try to send out the `Change View Request` message. Instead the `Recovery` message will be sent. And if the delegate has not sent out the `Commit message`, it will sent out the `Change View Request` message and move into a new view since it receives more than 2/3 `validator's Change View Request` messages. However, if the `speaker` sends out different `Prepare Request` messages, it is able to halt the consensus process.

Suppose there are 7 validators, A, B, C, D, E, F and G. A is the current speaker. In the following, we detail the steps for A to halt the consensus:

1. A makes 2 `prepare requests`, $P1$ and $P2$, with hashes of 2 different sets of transactions.

2. A sends $P1$ to B, C, D and E, $P2$ to F and G.

3. All non-speaker validators will validate the `prepare request` they received respectively and send out `prepare response`.

4. The `prepare responses` of B, C, D, E will not be accepted by F, G and vice versa. However, B, C, D and E will proceed to commit stage while F, G will request change view as no enough `prepare response` messages are received.

5. A will not send commit message. By doing so B, C, D, E will be in `commit` stage forever while F, G keeps trying to change view. Consensus halts.

Recommendation Make penalty mechanism for misbehaving committee members.

Status This issue has been confirmed.

4 Conclusion

In this security audit, we have analyzed the Neo3 Blockchain design and implementation. `Neo3` is the latest version of the `Neo` network and will be integrated with the `Poly Network` to allow for heterogeneous interoperability with other blockchains. Our audit has uncovered a list of 9 potential issues, and some of them involve unusual interactions among multiple modules.

Our journey through this audit is that the Neo3 Blockchain software is neatly organized and elegantly implemented and those identified issues are promptly confirmed and fixed. We would ike to commend the team for a well-done software project, and for quickly fixing reported issues. Also, as expressed in Section 1.4, we appreciate any constructive feedback or suggestions about this report.

References

[1] PeckShield. PeckShield Inc. https://www.peckshield.com.

[2] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[3] Lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[4] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html.