// Security Assessment

09.10.2025 - 11.04.2025

VerifiedX Core

VerifiedX

HALBORN

VerifiedX Core - VerifiedX

Prepared by: HALBORN

Last Updated 11/06/2025

Date of Engagement: September 10th, 2025 - November 4th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
68	5	9	31	17	6

TABLE OF CONTENTS

1. Introduction

The security review was commissioned by VerifiedX and was performed by Halborn security engineers. The broad scope was defined as an L1 and related consensus code review of the VerifiedX-Core repository and related services, including assessment of legacy and active consensus paths, P2P services, cryptographic primitives, and node/validator networking. The purpose of the engagement was to identify security defects and recommend mitigations to harden consensus, networking, cryptography, and state-handling components.

2. Assessment Summary

The engagement required multiple specialist reviews and took place over the period captured in the supplied findings of 40 days. A cross-functional Halborn team was applied and manual review was emphasized alongside automated scans and unit-test verification. The principal goals were detection of cryptographic, consensus, networking, and input-validation weaknesses and validation of remediations. The overall security posture of the codebase was strong after remediation activity: most issues flagged were fixed, legacy attack surface was removed, and multiple defensive controls were implemented. The most important fixes or improvements identified and confirmed as solved were:

- Cryptography: ECDSA signing/verification was hardened (zero-component retries and public-key curve membership checks).
- Consensus safety: V4 winner-selection determinism, parent-hash binding, fork-choice rules, and VRF tie/endian handling observations were addressed or documented.
- Networking and authentication: Signed consensus metadata, nonce-based handshake protection, replay prevention, and address–publicKey binding were implemented.
- Input validation and DoS hardening: JSON size/depth limits, safe parsing (TryParse), rate limiting, SignalRQueue global caps, and pre-validation checks for blocks and proofs were applied.
- Legacy code removal: Deprecated consensus paths and unused methods that exposed theoretical risks were removed or guarded, reducing attack surface.

A consolidated remediation state of "Solved" was reported for all of the findings in the provided dataset.

3. Test Approach And Methodology

The assessment was executed by sequencing discovery, targeted manual review, and automated analysis. Initial repository reconnaissance and scoping was performed to identify active execution paths versus legacy/unused code. Manual code review was then applied to high-risk components (consensus, cryptography, P2P servers, and state application). Automated static analysis and unit-test review was used to surface parsing errors, unsafe APIs, and deserialization risks. A verification phase was performed where developer-supplied remediation comments, commits, and unit test results were examined to confirm fixes.

The phases were as follows:

- Research and scoping: repository mapping and identification of active versus legacy code paths.
- Manual secure-code review: focused inspection of consensus algorithms, ECDSA/Elliptic Curve handling, message parsing, handshake logic, and P2P endpoints.
- Automated scans and tooling: static analyzers and JSON/serialization safety checks (details in Automated Testing section).
- Remediation verification: confirmation of applied fixes via code comments, commit references, and unit-test evidence when provided.

A balance was maintained between manual and automated work: manual review was prioritized for design-level consensus and cryptography issues while automated checks were used to validate input-parsing, deserialization, and potential DoS vectors. Confidence in coverage was increased by cross-validating manual findings with remediation evidence and unit tests provided in the context data.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: VerifiedX-Core

(b) Assessed Commit ID: dd57121

(c) Items in scope:

- Scope Overview
- Primary Directories & Files
- P2P/ (8,000+ lines)
- P2P networking
- Consensus protocols
- Services/
- BlockValidator* (1,500+ lines)
- - Block validation logic
- TransactionValidator* (2,100+ lines)
- - Transaction validation
- Consensus*
- - Consensus protocol implementation
- Data/
- StateData.cs (2,200+ lines)
- - State management
- BlockchainData.cs
- - Blockchain data operations
- Nodes/ (4,000+ lines)
- Node processing logic
- Models/
- Block.cs
- Blockchain.cs
- - Core blockchain models

REMEDIATION COMMIT ID:

- https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/8
- 71f203a
- a69709e
- f0e8bb1
- 33eb0ec

- 3b2521e
- NA
- d7cb9b5
- c8f095d
- 2dc90e7
- c8fd239
- 4b50cee
- f0e7907
- f684afb
- c22923d
- ee3471f
- 9403c34
- fe79406
- 533df68
- bcd88da
- 1a2e584
- 2c277f1
- 6f2ad36
- cfa8fb5
- 61c7b0e
- 85234d2
- 49a5635
- f71fee7
- f70ab47
- 46edb71
- 667066a
- 2985f08
- 156b8c1
- bfdf82b
- 06e138b
- 66dfc00
- 5c8fc56
- 842485e
- 4f5784a
- 3684968
- https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/10
- https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/11
- 0e3ffdb
- 1dfd18e
- 1a29d87
- aab9e83
- f35e354
- 7ecfc83
- 1de47fb

- 0b0134b
- e29ea79
- 7ecfc83
- bec76a1
- 654f7fa
- f8df1aa
- https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/9/commits/d82893b9c474730bb4ea25bb4de75bf0133441bd
- e2f16b6
- ccf1393
- 83f8ed3
- cb096dc

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	9	31	17	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
POINT VALIDATION LOGIC RETURNS INVERTED RESULTS	CRITICAL	SOLVED - 09/25/2025
RANDOMNESS MANIPULATION VIA LAST-REVEALER DUE TO LINEAR COMBINERANDOMS AND WEAK COMMIT/REVEAL	CRITICAL	SOLVED - 11/02/2025
BLOCK VALIDATION COUNTS RETIRED SIGNERS TOWARD MAJORITY WHILE THRESHOLD USES CURRENT SIGNERS	CRITICAL	SOLVED - 11/02/2025
FOREIGN PARENT ACCEPTANCE IN PROOF VALIDATION	CRITICAL	SOLVED - 11/02/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING FORK-CHOICE POLICY ENABLES PERSISTENT DIVERGENCE	CRITICAL	SOLVED - 11/02/2025
UNAUTHENTICATED CONSENSUS METADATA PARSING ALLOWS STATE MANIPULATION	HIGH	SOLVED - 10/04/2025
UNAUTHENTICATED PEER STATE UPDATES ALLOW LIVENESS DISRUPTION	HIGH	SOLVED - 10/25/2025
UNSIGNED CONSENSUS METADATA ALLOWS CLIENT STATE MANIPULATION	HIGH	SOLVED - 10/25/2025
MISSING SIGNATURE VERIFICATION IN RECEIVEDOWNLOADREQUEST	HIGH	SOLVED - 10/26/2025
MESSAGE SIGNATURE CHECK BYPASS FOR METHODCODE=0	HIGH	SOLVED - 11/02/2025
QUORUM COMPUTED FROM REGISTERED SIGNERS BUT WAITS USE LIVENESS	HIGH	SOLVED - 11/02/2025
DYNAMIC MEMBERSHIP WITHOUT PER-ROUND SNAPSHOT (SHIFTING QUORUM THRESHOLD)	HIGH	SOLVED - 11/02/2025
MISSING ANTI-EQUIVOCATION DETECTION ENABLES DOUBLE-VOTING	HIGH	SOLVED - 11/02/2025
NONDETERMINISTIC TIE-HANDLING IN VRF SELECTION	HIGH	SOLVED - 11/02/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNRESTRICTED DESERIALIZATION OF INCOMING PROOF LISTS	MEDIUM	SOLVED - 10/04/2025
HEIGHT CALCULATION USES INCORRECT PEER COLLECTION AFTER VALIDATOR CONNECTIVITY CHECK	MEDIUM	SOLVED - 10/04/2025
AUTHENTICATION HANDSHAKE VULNERABLE TO REPLAY ATTACKS	MEDIUM	SOLVED - 10/04/2025
UNCAUGHT PARSING EXCEPTION ENABLES HANDSHAKE DENIAL OF SERVICE	MEDIUM	SOLVED - 10/04/2025
UNAUTHENTICATED VALIDATOR DISCOVERY ENABLES NETWORK TOPOLOGY MANIPULATION	MEDIUM	SOLVED - 10/04/2025
UNSAFE STRING SLICING ON UNTRUSTED WALLET VERSION CAUSES CONNECTION FAILURES	MEDIUM	SOLVED - 10/04/2025
ADDRESS-BASED BANS APPLIED BEFORE AUTHENTICATION	MEDIUM	SOLVED - 10/04/2025
UNBOUNDED FIELD LENGTHS IN HANDSHAKE AND VALIDATOR MODELS	MEDIUM	SOLVED - 10/04/2025
FIRE-AND-FORGET BROADCASTS WITHOUT ERROR HANDLING OR FLOW CONTROL	MEDIUM	SOLVED - 10/05/2025
MISSING VALIDATOR ROLE ASSERTION IN BLOCK RECEPTION	MEDIUM	SOLVED - 10/05/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SYNCHRONOUS REMOTE PORT CHECK IN HANDSHAKE PATH	MEDIUM	SOLVED - 10/25/2025
MISSING ADDRESS–PUBLICKEY BINDING IN VALIDATOR HANDSHAKE	MEDIUM	SOLVED - 10/25/2025
SILENT EXCEPTION HANDLING HIDES ABUSE AND OPERATIONAL FAULTS	MEDIUM	SOLVED - 10/25/2025
UNBOUNDED WINNER LIST RESPONSES WITHOUT PAGINATION OR SIZE LIMITS	MEDIUM	SOLVED - 10/25/2025
BLOCK BROADCAST NOT GATED ON VALIDATION SUCCESS	MEDIUM	SOLVED - 10/25/2025
EXCESSIVE PARALLEL REQUESTS AND STATIC BACKOFF	MEDIUM	SOLVED - 10/25/2025
UNSAFE RESPONSE PARSING RISKS EXCEPTIONS AND DESYNCHRONIZATION	MEDIUM	SOLVED - 10/25/2025
UNCHECKED SPLIT-BASED PARSING IN MESSAGE/HASH ENDPOINTS	MEDIUM	SOLVED - 10/25/2025
UNBOUNDED TRANSACTION BROADCAST LIST INGESTION	MEDIUM	SOLVED - 10/26/2025
UNSAFE TIMESTAMP PARSING AND MISSING NONCE IN HANDSHAKE	MEDIUM	SOLVED - 10/25/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INVERTED DUPLICATE HANDLING LOGIC IN TASK ANSWER PROCESSING	MEDIUM	SOLVED - 10/25/2025
IP-ONLY GATING ENABLES SESSION HIJACK AND MISROUTING	MEDIUM	SOLVED - 10/25/2025
UNSAFE TIMESTAMP PARSING AND MISSING NONCE IN BLOCKCASTER HANDSHAKE	MEDIUM	SOLVED - 10/25/2025
RECEIVEBLOCKVAL HEAVY PATH WITHOUT SIGNALRQUEUE/DOS GUARD	MEDIUM	SOLVED - 10/26/2025
AGGRESSIVE PARALLELISM/BACKOFF IN PEER CONNECTIONS AND UPDATES	MEDIUM	SOLVED - 10/25/2025
IP-KEYED SESSION MAPPING ENABLES HIJACK/MISDIRECTION	MEDIUM	SOLVED - 10/25/2025
DOS THROTTLING WEAKNESSES IN SIGNALRQUEUE	MEDIUM	SOLVED - 10/25/2025
TRANSACTION NONCE ORDERING NOT ENFORCED	MEDIUM	SOLVED - 11/02/2025
RESERVE CALLBACK/RECOVER LACK IDEMPOTENCE; LOCKED BALANCE UNDERFLOW RISK	MEDIUM	SOLVED - 11/04/2025
MISSING FEE FLOOR AND GLOBAL MEMPOOL LIMITS ENABLE ECONOMIC/SPACE DOS	MEDIUM	SOLVED - 11/04/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNBOUNDED DESERIALIZATION AND MISSING CANCELLATION/BACKPRESSURE IN NODEDATAPROCESSOR	MEDIUM	SOLVED - 11/04/2025
SIGNATURE GENERATION LACKS VALIDATION FOR ZERO COMPONENTS	LOW	SOLVED - 09/25/2025
SIGNATURE VERIFICATION ACCEPTS OFF-CURVE PUBLIC KEYS	LOW	SOLVED - 09/25/2025
SYNCHRONOUS DISPOSAL BLOCKS ON ASYNCHRONOUS OPERATIONS CAUSING POTENTIAL HANGS	LOW	SOLVED - 10/04/2025
MISSING DOS GUARD AND RATE LIMITS ON BLOCK RECEPTION	LOW	SOLVED - 10/05/2025
MISSING PRE-VALIDATION FILTERS ON BLOCK RECEPTION	LOW	SOLVED - 10/05/2025
PREMATURE EXIT FROM MAJORITY CALCULATION	LOW	SOLVED - 10/25/2025
SYNCHRONOUS BLOCKING ON ASYNC DISPOSAL	LOW	SOLVED - 10/25/2025
UNBOUNDED GROWTH OF MESSAGE/HASH CACHES	LOW	SOLVED - 10/25/2025
PRE-AUTHENTICATION STATE AND BALANCE VALIDATION	LOW	SOLVED - 10/25/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SIGNATURE REUSE MAP GROWS WITHOUT CLEANUP	LOW	SOLVED - 10/25/2025
UNSAFE PARSING OPERATIONS IN TASK ANSWER PROCESSING	LOW	SOLVED - 10/25/2025
UNBOUNDED JSON INPUTS AND UNSAFE ASSET NAME HANDLING	LOW	SOLVED - 10/25/2025
BLOCKCASTER HANDSHAKE: PRE-AUTHENTICATION STATE/BALANCE CHECKS	LOW	SOLVED - 10/25/2025
UNSAFE PARSING AND SUBSTRING OOB	LOW	SOLVED - 10/25/2025
PAYLOAD FIELDS NOT CRYPTOGRAPHICALLY BOUND TO AUTHENTICATED IDENTITY	LOW	SOLVED - 10/25/2025
TOCTTOU IN PER-IP QUEUE ACCOUNTING (CONNECTIONCOUNT/BUFFERCOST)	LOW	SOLVED - 10/25/2025
TRANSACTION STALENESS CHECK DEPENDS ON DOWNLOAD STATE	LOW	SOLVED - 11/02/2025
DOCUMENTATION CONTAINS SPELLING ERRORS AND MISLEADING DESCRIPTIONS	INFORMATIONAL	SOLVED - 09/25/2025
HARDCODED TIMEOUTS AND FIXED DELAYS WITHOUT OBSERVABILITY	INFORMATIONAL	SOLVED - 10/05/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNAUTHENTICATED VALIDATOR LIST UPDATES AND WEAK BINDING CHECKS	INFORMATIONAL	SOLVED - 10/25/2025
UNBOUNDED VALIDATOR REGISTRY GROWTH WITHOUT PRUNING OR TTL	INFORMATIONAL	SOLVED - 10/25/2025
VRFNUMBER ENDIANNESS DEPENDENCY CAN CAUSE CROSS-PLATFORM CONSENSUS SPLITS	INFORMATIONAL	SOLVED - 11/03/2025
V4 PROOF VALIDATION LACKS COMMITTEE MEMBERSHIP AND WINNER ENFORCEMENT	INFORMATIONAL	SOLVED - 11/02/2025

7. FINDINGS & TECH DETAILS

7.1 POINT VALIDATION LOGIC RETURNS INVERTED RESULTS

// CRITICAL

Description

The `contains` method in the `CurveFp` class has inverted logic that incorrectly validates whether a point lies on the elliptic curve. When the elliptic curve equation `y² = x³ + ax + b (mod p)` is satisfied (meaning the point is valid), the method returns `false`. Conversely, when the equation is not satisfied (meaning the point is invalid), the method returns `true`. This occurs because the method returns `false` when `Integer.modulo(BigInteger.Pow(p.y, 2) - (BigInteger.Pow(p.x, 3) + A * p.x + B), P).IsZero` evaluates to `true`, which indicates the point satisfies the curve equation and should be considered valid. The base point `G` of secp256k1, which is mathematically guaranteed to be on the curve, incorrectly returns `false` when passed to the `contains` method. Similarly, the origin point `(0,0)`, which does not lie on the secp256k1 curve, incorrectly returns `true`.

Proof of Concept

```
using System;
using System.Numerics;
using ReserveBlockCore.EllipticCurve;

class WrongContains
{
    static void Main(string[] args)
    {
        CurveFp secp256k1 = Curves.getCurveByName("secp256k1");
        // Base point (on-curve)
        Point P = secp256k1.G;
        bool containsP = secp256k1.contains(P);
        Console.WriteLine($"Curve contains base point: {containsP}"); // Expected: true, Bug: false


        // Off-curve point
        Point Q = new Point(BigInteger.Zero, BigInteger.Zero);
        bool containsQ = secp256k1.contains(Q);
        Console.WriteLine($"Curve contains off-curve point: {containsQ}"); // Expected: false, Bug: true
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (10.0)

Recommendation

Invert the return values in the `contains` method. When the modulo operation results in zero, indicating the point satisfies the elliptic curve equation, return `true`. When the modulo operation is non-zero, indicating the point does not satisfy the equation, return `false`. The corrected logic should return `true` when

`Integer.modulo(BigInteger.Pow(p.y, 2) - (BigInteger.Pow(p.x, 3) + A * p.x + B), P).IsZero` and `false` otherwise.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue in https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/8.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/8

7.2 RANDOMNESS MANIPULATION VIA LAST-REVEALER DUE TO LINEAR COMBINERANDOMS AND WEAK COMMIT/REVEAL

// CRITICAL

Description

The consensus randomness generation uses a linear combination function that enables last-revealer attacks through deterministic manipulation of the final random value. The `CombineRandoms` method computes the modular sum of all revealed validator answers, allowing the final participant to calculate the exact value needed to force any desired outcome after observing other reveals. Combined with the lack of strict anti-equivocation controls and soft reveal windows, an adaptive attacker can delay their reveal, compute `answer_last = target - sum(others) mod range`, and deterministically steer candidate selection since winner ranking is based on proximity to the chosen random value.

Proof of Concept

```
public static int CombineRandoms(IList<int> randoms, int minValue, int maxValue)
{
    return Modulo(randoms.Sum(x => (long)x), maxValue - minValue) + minValue;
}

var EncryptedAnswers = await ConsensusClient.ConsensusRun(0, MyEncryptedAnswer, MyEncryptedAnswerSignature, 2000,
// ... later
var DecryptedAnswers = await ConsensusClient.ConsensusRun(2, MyDecryptedAnswer, MyEncryptedAnswer, 2000, RunType.

var Answers = DecryptedAnswers.Select(x =>
{
    var split = x.Message.Split(':');
    var encryptedAnswer = EncryptedAnswers.Where(y => y.Address == x.Address).FirstOrDefault();
    if (split.Length != 2 || long.Parse(split[0]) != Height || encryptedAnswer.Address == null ||
        !SignatureService.VerifySignature(x.Address, x.Message, encryptedAnswer.Message))
        return -1;
    return int.Parse(split[1]);
}).Where(x => x != -1).ToArray();
var ChosenAnswer = CombineRandoms(Answers, 0, int.MaxValue);

var PotentialWinners = ValidSubmissions
    .Where(x => !BadIPs.Contains(x.IPAddress) && !BadAddresses.Contains(x.RBXAddress))
    .GroupBy(x => x.Answer)
    .Where(x => x.Count() == 1)
    .Select(x => x.First())
    .OrderBy(x => Math.Abs(x.Answer - ChosenAnswer))
    .ThenBy(x => x.Answer)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (10.0)

Recommendation

• Replace the linear sum aggregation with a proper commit-reveal scheme that binds each validator to a per-round secret commitment using cryptographic hashes before any reveals occur.

• Implement per-validator commit verification that rejects reveals not matching their initial commitments, preventing adaptive answer selection based on observed values.
• Establish fixed reveal windows with penalties for late submissions and consider non-malleable aggregation methods such as XOR of uniformly distributed secrets to eliminate manipulation opportunities.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/71f203a1c099e9485aab88a016b89af3867be5b5

7.3 BLOCK VALIDATION COUNTS RETIRED SIGNERS TOWARD MAJORITY WHILE THRESHOLD USES CURRENT SIGNERS

// CRITICAL

Description

The Version 3 block validation logic contains a critical threshold mismatch that enables consensus bypass through retired signer exploitation. The `Version3Rules` method accepts signatures from both current signers (`Globals.Signers`) and retired signers (`Globals.RetiredSigners`) when validating blocks, but calculates the required majority threshold using only the current signer count via `Signer.Majority()` . This discrepancy allows attackers with access to retired private keys to construct signature sets that meet the majority-of-current threshold without any participation from legitimate current committee members, enabling unauthorized block finalization and potential chain takeover.

Proof of Concept

```
foreach (var AddressSignature in AddressSignatures)
{
    var split = AddressSignature.Split(':');
    var (Address, Signature) = (split[0], split[1]);
    if (!Globals.Signers.ContainsKey(Address) && !Globals.RetiredSigners.ContainsKey(Address))
        return (false, "Signers Did Not Have Key.");
    if (!(SignatureService.VerifySignature(Address, block.Hash, Signature)))
        return (false, "Signature Failed to verify");
    ValidCount++;
    Addresses.Add(Address);
}
if (ValidCount == Addresses.Count && ValidCount >= Signer.Majority())
    return (true, "");

public static int Majority()
{
    return NumSigners() / 2 + 1;
}
public static int NumSigners()
{
    return Globals.Signers.Count;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (10.0)

Recommendation

• Restrict signature validation to only count signatures from the current signer set for threshold calculation, completely excluding retired signers from consensus participation.
• Implement explicit epoch-based validation that binds blocks to specific committee snapshots with defined validity windows to prevent cross-epoch signature replay.
• Add signer-set hash validation to the signed payload to ensure blocks cannot be validated against different committee compositions than intended by the original signers.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/a69709ec67fd4f3906ffc737ae51a7ef8c2fb9ae

7.4 FOREIGN PARENT ACCEPTANCE IN PROOF VALIDATION

// CRITICAL

Description

The V4 winner selection mechanism in `SortProofs` validates candidate proofs using their embedded `PreviousBlockHash` without verifying it matches the local node's current tip at `Globals.LastBlock.Hash`. This allows nodes to accept and rank proofs that target different parent blocks than their own chain state. When nodes operate on different chain tips due to network partitions or propagation delays, they can select different winners at the same height by accepting proofs built for foreign parents, leading to consensus divergence and persistent forks that compound other V4 vulnerabilities.

Proof of Concept

```
public static async Task<Proof?> SortProofs(List<Proof> proofs, bool isWinnerList = false)
{
    try
    {
        var processHeight = Globals.LastBlock.Height + 1;

        var validProofs = proofs.Where(p =>
            p.BlockHeight == processHeight &&
            p.VerifyProof() &&
            !Globals.ABL.Exists(x => x == p.Address)
        ).ToList();

        // Sort deterministically by VRF number
        return validProofs
            .OrderBy(x => x.VRFNumber)  // Closest to zero wins
            .FirstOrDefault();

public bool VerifyProof()
{
    try
    {
        var proofResult = ProofUtility.VerifyProof(PublicKey, BlockHeight, PreviousBlockHash, ProofHash);
        return proofResult;
    }
    catch { return false; }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (10.0)

Recommendation

- Enforce parent hash validation by requiring `p.PreviousBlockHash == Globals.LastBlock.Hash` in `SortProofs` before accepting proofs for winner selection processing.
- Alternatively, modify `VerifyProof()` to accept the expected parent hash as a parameter and ignore the `PreviousBlockHash` field carried within proof objects to prevent foreign parent acceptance.
- Integrate with comprehensive V4 security improvements including deterministic tie-breaking, committee membership validation, block-to-tip binding, and endianness normalization to ensure robust consensus

operation.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue by adding a validation check in ProofUtility.SortProofs method to verify that proof PreviousBlockHash matches local nodes current chain tip at Globals.LastBlock.Hash.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f0e8bb1097c29cd64b18bd2bcbc5f5045c395e7d

7.5 MISSING FORK-CHOICE POLICY ENABLES PERSISTENT DIVERGENCE

// CRITICAL

Description

The blockchain implementation lacks an explicit fork-choice policy to handle competing valid blocks at the same height, relying instead on first-seen acceptance without reorg capabilities. When multiple valid blocks pass stateless validation for `LastBlock.Height + 1`, nodes can permanently lock onto different branches with no mechanism to reconcile to a canonical chain. The `BlockValidatorService.ValidateBlock` accepts any qualifying candidate and immediately updates `Globals.LastBlock`, while the receive path queues blocks per height without selection rules for competing blocks. This creates persistent forks during network races or adversarial conditions where different nodes observe different block orderings.

Proof of Concept

```
// Validation entrypoint accepts any block matching height+1 after local checks
var result = await BlockValidatorService.ValidateBlock(block, ...);
if (result)
{
    await BlockchainData.AddBlock(block, updateCLI);//add block to chain.
    // Globals.LastBlock updated, no fork-choice across competing candidates
}

// Receive path queues blocks per height; no selection rule across competing blocks
if (currentHeight >= nextHeight && BlockDownloadService.BlockDict.TryAdd(currentHeight, (nextBlock, IP)))
{
    if (Globals.LastBlock.Height < nextBlock.Height)
        await BlockValidatorService.ValidateBlocks();
    if (nextHeight < currentHeight)
        await BlockDownloadService.GetAllBlocks();
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (10.0)

Recommendation

• Define and implement a canonical fork-choice rule such as finalized adjudicator-majority signature weight verification or explicit longest/heaviest chain selection to resolve competing branches deterministically.
• Develop branch management capabilities to maintain competing block headers with minimal state tracking and implement automatic reorg functionality when the preferred branch becomes evident.
• Establish persistent validation and storage of consensus justifications including adjudicator signatures for V3 consensus to drive reorg decisions and provide cryptographic finality guarantees.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/33eb0eccf23f7dd5febaa61a79e617d5b5ef9a8f

7.6 UNAUTHENTICATED CONSENSUS METADATA PARSING ALLOWS STATE MANIPULATION

// HIGH

Description

The `P2PValidatorClient` class contains a critical security flaw in its `UpdateMethodCode` method where consensus coordination metadata received from remote validator peers is processed without authentication or proper input validation. The method invokes `RequestMethodCode` on peer connections and expects a colon-delimited response string in the format `height:method:finalized`. However, the implementation directly splits this response using `Response.Split(':')` and immediately parses the resulting array elements with `long.Parse(remoteMethodCode[0])`, `int.Parse(remoteMethodCode[1])`, and boolean conversion `remoteMethodCode[2] == "1"` without verifying the array length or validating the parsed values. This creates multiple attack vectors where malicious validator peers or man-in-the-middle attackers can manipulate critical consensus state variables including `NodeHeight`, `MethodCode`, and `IsFinalized` flags. While the actual block payloads and transaction data remain cryptographically protected through existing signature verification, this unauthenticated coordination metadata directly affects consensus timing decisions, validator selection logic, cache management through `ConsensusServer.RemoveStaleCache(node)`, and overall network synchronization behavior.

The vulnerability is particularly concerning because it operates within the validator consensus network where nodes must coordinate block production timing and finalization status. Attackers can exploit this to cause localized denial of service through parsing exceptions when sending malformed responses with insufficient colon-separated segments, manipulate consensus progress by injecting false height or method code values, or disrupt cache coherency by triggering inappropriate cache removals. The transport layer uses HTTP/WebSockets with `RemoteCertificateValidationCallback = (sender, cert, chain, errors) => true`, which disables certificate validation and further enables man-in-the-middle attacks on the coordination metadata.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

Implement comprehensive authentication and validation for consensus coordination metadata by modifying the `RequestMethodCode` endpoint to include the `height:method:finalized` values within the cryptographically signed response payload structure, ensuring these critical coordination values receive the same cryptographic protection as block data. Add robust input validation by checking that `Response.Split(':')` returns exactly 3 elements before array access, replace unsafe parsing methods with `long.TryParse` and `int.TryParse` that include reasonable bounds validation for height values and method codes, and implement proper error handling that logs malformed responses at a controlled rate while applying exponential backoff to peers sending invalid data. Strengthen transport security by enabling proper certificate validation in the WebSocket configuration and consider implementing mutual TLS authentication

for validator-to-validator communication to provide defense-in-depth against network-level attacks on consensus coordination traffic.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/3b2521e34f04a029bcab22f17101c8f028c4089c

7.7 UNAUTHENTICATED PEER STATE UPDATES ALLOW LIVENESS DISRUPTION

// HIGH

Description

In the consensus contract, peer state fields such as height, method code, and finalized status are updated from unauthenticated response prefixes before any signature verification occurs. This state is then used to control consensus round cancellation. A malicious peer can forge these prefix values to repeatedly trigger early cancellation, preventing the node from reaching consensus and potentially stalling the chain locally or across multiple nodes if collusion occurs. Signature checks only apply to the trailing message or hash body, not the prefix fields, leaving the consensus process vulnerable to liveness denial-of-service and coordination skew.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
if (Response != null)
{
    var PrefixSplit = Response.Split(new[] { '|' }, 2);
    var Prefix = PrefixSplit[0].Split(':');
    if (Now > peer.LastMethodCodeTime)
    {
        lock (ConsensusServer.UpdateNodeLock)
        {
            peer.LastMethodCodeTime = Now;
            peer.NodeHeight = long.Parse(Prefix[0]);
            peer.MethodCode = int.Parse(Prefix[1]);
            peer.IsFinalized = Prefix[2] == "1";
        }
        ConsensusServer.RemoveStaleCache(peer);
    }
}
...
if (peer.NodeHeight > currentHeight)
{
    cts.Cancel();
    break;
}
```

Signature verification is only performed on the trailing body, not the prefix:

```
if (PrefixSplit.Length == 2)
{
    var arr = PrefixSplit[1].Split("::;");
    var (address, message, signature) = (arr[0], arr[1].Replace("::", ":"), arr[2]);
    if (SignatureService.VerifySignature(address, message, signature) && ConsensusServer.GetState().Status == Cons
        messages[address] = (message, signature);
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

- Include prefix fields (height, method code, finalized flag, timestamp/nonce) inside the signed payload and verify them before updating peer state.
- Alternatively, ignore prefix values from unauthenticated peers or require a per-peer session MAC to bind state changes to authenticated sources.
- Add sanity checks and monotonicity rules, such as bounding the jump in height or method code across heartbeats, before allowing consensus round cancellation.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.8 UNSIGNED CONSENSUS METADATA ALLOWS CLIENT STATE MANIPULATION

// HIGH

Description

In the consensus contract, the server emits an unsigned prefix containing height, method code, and finalized status, which clients use to update peer state and control consensus flow. Because this prefix is not cryptographically bound to the responder identity, any reachable peer can send manipulated values, allowing malicious or misbehaving nodes to influence client consensus state. This can result in premature round cancellation or advancement, causing liveness issues and disrupting consensus reliability.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
public string RequestMethodCode(long height, int methodCode, bool isFinalized)
{
    var ip = GetIP(Context);
    ...
    UpdateNode(node, height, methodCode, isFinalized);

    UpdateConsensusDump(ip, "RequestMethodCode", height + " " + methodCode + " " + isFinalized, (Globals.LastBlocl
        (ConsenusStateSingelton.Status == ConsensusStatus.Finalized ? 1 : 0));
    return (Globals.LastBlock.Height).ToString() + ":" + ConsenusStateSingelton.MethodCode + ":" +
        (ConsenusStateSingelton.Status == ConsensusStatus.Finalized ? 1 : 0);
}
```

Other endpoints similarly construct and return unsigned prefixes:

```
UpdateNode(node, height - 1, methodCode, false);
Prefix = (Globals.LastBlock.Height).ToString() + ":" + ConsenusStateSingelton.MethodCode + ":" +
                    (ConsenusStateSingelton.Status == ConsensusStatus.Finalized ? 1 : 0);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:M/A:M/I:M/D:N/Y:N (7.5)

Recommendation

- Bind consensus metadata (height, method code, finalized status) to the responder identity using cryptographic signatures.
- Require clients to verify the authenticity of consensus metadata before updating peer state or controlling consensus flow.
- Reject unauthenticated or unsigned prefixes and treat them as benign no-ops, logging metrics for monitoring.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.9 MISSING SIGNATURE VERIFICATION IN RECEIVEDOWNLOADREQUEST

// HIGH

Description

The `ReceiveDownloadRequest` method in the beacon download coordination system accepts asset transfer requests without performing signature verification. The signature validation logic using `SignatureService.VerifySignature` is present in the code but has been commented out, completely bypassing authentication checks. This allows any reachable peer to forge download requests by simply referencing existing smart contract UIDs and assets without proving ownership or authorization. The method retrieves smart contract state via `SmartContractStateTrei.GetSmartContractState` but fails to validate that the requester has legitimate access rights to initiate asset transfers.

Proof of Concept

```
public async Task<bool> ReceiveDownloadRequest(BeaconData.BeaconDownloadData bdd)
{
    //return await SignalRQueue(Context, 1024, async () =>
    //{
        bool result = false;
        var peerIP = GetIP(Context);
        var beaconPool = Globals.BeaconPool.Values.ToList();
        try
        {
            if (bdd != null)
            {
                var scState = SmartContractStateTrei.GetSmartContractState(bdd.SmartContractUID);
                if (scState == null)
                {
                    return result; //fail
                }

                //var sigCheck = SignatureService.VerifySignature(scState, bdd.SmartContractUID, bdd.Signature);
                //if (sigCheck == false)
                //{
                //    return result; //fail
                //}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

• Reinstate and enforce signature verification by uncommenting and properly implementing the `SignatureService.VerifySignature` check against the rightful owner or designated next owner before processing any download requests.
• Implement strong binding between IP addresses and authenticated identities, ensuring that beacon references are cryptographically tied to verified user credentials and rejecting requests that fail authentication.

• Add comprehensive rate limiting mechanisms and audit logging for failed authentication attempts to detect and mitigate potential abuse patterns while maintaining security visibility.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/d7cb9b512ceb170680435639f17f3f91b7b53d0b

7.10 MESSAGE SIGNATURE CHECK BYPASS FOR METHODCODE=0

// HIGH

Description

The `ConsensusServer.Message` method contains a critical operator precedence vulnerability that allows unauthenticated message insertion for `methodCode == 0`. Due to C# operator precedence where `&&` binds tighter than `||`, the conditional expression creates two distinct evaluation branches: the left branch `(message != null && (methodCode == 0 && state.MethodCode != 0))` bypasses both signature verification and height constraints, while only the right branch enforces proper authentication. This allows attackers to inject arbitrary messages into the server-side cache at any height without verification, creating denial-of-service risks through memory exhaustion and cache pollution that can impact consensus liveness.

Proof of Concept

```
if (message != null && ((methodCode == 0 && state.MethodCode != 0)) ||
    (height == Globals.LastBlock.Height + 1
    && ((methodCode == state.MethodCode && state.Status != ConsensusStatus.Finalized) ||
        (methodCode == state.MethodCode + 1 && state.Status == ConsensusStatus.Finalized)))
    && SignatureService.VerifySignature(node.Address, message, signature))
    messages[node.Address] = (message, signature);

var MessageKeysToKeep = ConsensusServer.Messages.Where(x => (x.Key.Height == Height && x.Key.MethodCode == methodC
    (x.Key.Height > Height && x.Key.MethodCode == 0))
    .Select(x => x.Key).ToHashSet();
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

• Fix the operator precedence issue by adding explicit parentheses to ensure signature verification and height validation are required for all code paths, including `methodCode == 0` scenarios.
• Implement per-address rate limiting and bounded cache sizes for each `(height, methodCode)` bucket to prevent memory exhaustion from malicious message flooding.
• Add comprehensive logging and metrics for unauthenticated message attempts and cache eviction events to improve visibility into potential attacks and system health monitoring.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/c8f095d60b3905c1a70bed4e9ac923eacb8e80d8

7.11 QUORUM COMPUTED FROM REGISTERED SIGNERS BUT WAITS USE LIVENESS

// HIGH

Description

The consensus mechanism computes majority thresholds from the complete registered signer set while using recently active nodes for wait-set determination, creating a fundamental liveness mismatch. The system calculates `Majority = NumNodes / 2 + 1` from `Globals.Signers.Keys` but waits only for nodes meeting recent activity criteria through `LastMethodCodeTime` validation. When registered signers are offline or experiencing network issues, the consensus requires a majority that cannot be satisfied by the available live participants, causing frequent round failures and preventing finalization despite having sufficient active validators to reach consensus safely.

Proof of Concept

```
var CurrentAddresses = Globals.Signers.Keys.ToHashSet();
var NumNodes = CurrentAddresses.Count;
Majority = NumNodes / 2 + 1;

return Globals.Nodes.Values.Where(x => Now - x.LastMethodCodeTime < wait && ((x.NodeHeight + 2 == height && metho
    (x.NodeHeight + 1 == height && (x.MethodCode == methodCode || (x.MethodCode == methodCode - 1 && x.IsFinalize
    .Select(x => x.Address).ToHashSet();

var RemainingAddressCount = !ConsensusSource.IsCancellationRequested ? WaitForAddresses.Except(Messages.Select(x =
if ((methodCode != 0 && Messages.Count + RemainingAddressCount < Majority) ||
    (RemainingAddressCount == 0 && Messages.Count >= Majority))
    break;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

• Align majority threshold calculations with the same cohort used for wait-set determination, using recently active participants rather than the full registered signer set for quorum computation.
• Implement dynamic quorum adjustment mechanisms that can gracefully degrade when the active validator set shrinks below registered capacity, with explicit minimum thresholds for safety.
• Establish clear epoch-based membership rules with consistent snapshots that govern both quorum counting and wait-set determination throughout each consensus round.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/2dc90e70cdf6c724756c18876b124ba2233c00ef

7.12 DYNAMIC MEMBERSHIP WITHOUT PER-ROUND SNAPSHOT (SHIFTING QUORUM THRESHOLD)

// HIGH

Description

The consensus system recomputes signer membership and majority thresholds dynamically within consensus rounds rather than using immutable per-round snapshots. The `Globals.Signers` set and resulting `Majority` calculation can change mid-round due to operator actions, configuration reloads, or testnet membership updates via `Signer.UpdateSigningAddresses()`. This dynamic recalculation creates inconsistent participation expectations across nodes and can alter liveness characteristics unpredictably, as different nodes may disagree on the required majority threshold if their membership views diverge during active consensus rounds.

Proof of Concept

```
var CurrentAddresses = Globals.Signers.Keys.ToHashSet();
var NumNodes = CurrentAddresses.Count;
Majority = NumNodes / 2 + 1;

Signer.UpdateSigningAddresses() // updates Globals.Signers based on height and testnet flags
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

• Implement explicit epoch-based membership snapshots at the start of each consensus round and maintain consistent quorum thresholds throughout the entire round duration.
• Serialize all membership changes to take effect only at designated round boundaries, preventing mid-round threshold modifications that can cause inconsistent behavior.
• Add comprehensive logging and assertions to detect and prevent dynamic threshold changes during active consensus rounds, ensuring protocol safety guarantees are maintained.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/c8fd2393ada0536915930eae3a7bc23858f1a286

7.13 MISSING ANTI-EQUIVOCATION DETECTION ENABLES DOUBLE-VOTING

// HIGH

Description

The consensus server stores entries in `Messages` and `Hashes` dictionaries using a last-write-wins approach without detecting conflicting submissions from the same signer. At any given `(height, methodCode)`, new submissions from a signer overwrite previous ones without conflict detection, logging, or penalties. Malicious signers can exploit this by submitting different values to different peers or changing votes mid-round, creating divergent message sets across nodes. This prevents nodes from matching the same aggregated hash since the final hash computation depends on the complete message set, degrading consensus liveness and availability.

Proof of Concept

```
var messages = Messages.GetOrAdd((height, methodCode), new ConcurrentDictionary<string, (string Message, string S
// ...
if (message != null && /* phase/height branch */ && SignatureService.VerifySignature(node.Address, message, signat
    messages[node.Address] = (message, signature);

var hashes = Hashes.GetOrAdd((height, methodCode), new ConcurrentDictionary<string, (string Hash, string Signature
// ...
if (hash != null && /* phase/height branch */ && SignatureService.VerifySignature(node.Address, hash, signature))
    hashes[node.Address] = (hash, signature);

var FinalizedMessages = Messages.OrderBy(x => x.Key).ToArray();
var MyHash = Ecdsa.sha256(string.Join("", FinalizedMessages.Select(x => Ecdsa.sha256(x.Value.Message))));
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

• Implement first-write-wins enforcement per `(address, height, methodCode)` tuple and reject subsequent conflicting submissions with comprehensive logging of double-vote attempts.
• Establish penalty mechanisms for detected equivocation including rate limiting, reputation scoring, temporary banning, or protocol-level slashing to deter repeat offenders.
• Bind submissions explicitly to round identifiers including height, methodCode, and epoch parameters while maintaining persistent audit trails for forensic analysis of consensus attacks.
• Deploy monitoring systems with metrics and alerting capabilities to track equivocation attempts and identify their sources for network security analysis.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/4b50cee62fcba46918cef640d68c442e64e02dff

7.14 NONDETERMINISTIC TIE-HANDLING IN VRF SELECTION

// HIGH

Description

The V4 consensus mechanism selects block producers by sorting VRF proofs in ascending order by `VRFNumber` and selecting the first entry. When multiple validators generate identical 31-bit `VRFNumber` values, the absence of deterministic tiebreaking logic causes winner selection to depend on collection iteration order, which varies across different nodes. The `CreateProof` method generates VRF numbers in a 31-bit space using `(randomBytesAsInt & 0x7FFFFFFF)`, creating non-negligible collision probability that can result in different nodes selecting different winners at the same block height, leading to consensus forks.

Proof of Concept

```
var validProofs = proofs.Where(p =>
    p.BlockHeight == processHeight &&
    p.VerifyProof() &&
    !Globals.ABL.Exists(x => x == p.Address)
).ToList();

// Sort deterministically by VRF number
return validProofs
    .OrderBy(x => x.VRFNumber)  // Closest to zero wins
    .FirstOrDefault();

public static async Task<(uint, string)> CreateProof(string address, string publicKey, long blockHeight, string pr
{
    uint vrfNum = 0;
    var proof = "";
    string seed = publicKey + blockHeight.ToString() + prevBlockHash;
    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(seed));
        int randomBytesAsInt = BitConverter.ToInt32(hashBytes, 0);
        uint nonNegativeRandomNumber = (uint)(randomBytesAsInt & 0x7FFFFFFF);
        vrfNum = nonNegativeRandomNumber;   // 31-bit space → non-zero collision risk
        proof = ProofUtility.CalculateSHA256Hash(seed + vrfNum.ToString());
    }
    return (vrfNum, proof);
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

• Implement canonical ordering with explicit tiebreaking rules using `OrderBy(VRFNumber).ThenBy(ProofHash).ThenBy(PublicKey).ThenBy(Address)` to ensure deterministic selection across all nodes.

• Expand the VRF number space to use wider, endian-agnostic scalars such as UInt64 big-endian encoding or full 256-bit values to significantly reduce collision probability.

• Create comprehensive test suites to verify deterministic winner selection under colliding `VRFNumber` scenarios across different platforms and runtime environments.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f0e79072d7c340c0400ee277dea8592fb9707a36

7.15 UNRESTRICTED DESERIALIZATION OF INCOMING PROOF LISTS

// MEDIUM

Description

The contract `P2PValidatorServer` processes incoming JSON payloads for proofs and winner lists using unbounded deserialization in the `SendProofList` and `SendWinningProofList` functions. These methods use `JsonConvert.DeserializeObject<List<Proof>>(proofJson)` on untrusted input without enforcing limits on input size, JSON depth, or collection length, and do not validate the schema or structure of the deserialized objects. This allows attackers to submit excessively large or deeply nested JSON payloads, or inject malformed objects, leading to high CPU and memory consumption during parsing and processing. Such behavior can degrade node availability and liveness, especially if combined with network flooding or repeated requests. The `GetWinningProofList` function also serializes potentially large collections for outbound responses without bounding the size, which can further contribute to resource exhaustion.

Proof of Concept

```
// Unbounded deserialization of proof lists
public async Task<bool> SendProofList(string proofJson)
{
    var proofList = JsonConvert.DeserializeObject<List<Proof>>(proofJson);
    if (proofList?.Count() == 0) return false;
    if (proofList == null) return false;
    await ProofUtility.SortProofs(proofList);
    return true;
}
```

```
// Unbounded serialization of winner lists
public async Task<string> GetWinningProofList()
{
    string result = "0";
    if(Globals.WinningProofs.Count() != 0)
    {
        var heightMax = Globals.LastBlock.Height + 10;
        var list = Globals.WinningProofs.Where(x => x.Key <= heightMax).Select(x => x.Value).ToList();
        if(list != null)
            result = JsonConvert.SerializeObject(list);
    }
    return result;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:M/D:N/Y:N (6.3)

Recommendation

- Enforce strict input limits by capping the byte size, JSON depth, and collection length for all incoming payloads.

- Validate the schema and required fields of each `Proof` object before processing, and reject any payloads that exceed defined thresholds or contain invalid elements.
- Configure the JSON deserializer to disable polymorphic type handling and set a maximum depth.
- Prefer using explicit models and source-generated serializers where possible.
- Implement per-IP rate limiting and cooldowns for repeated invalid or oversized payloads.
- Consider batching or streaming for legitimate large lists to avoid materializing unbounded collections in memory.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f684afb03646e5264a7762628fa7833ff9b34445

7.16 HEIGHT CALCULATION USES INCORRECT PEER COLLECTION AFTER VALIDATOR CONNECTIVITY CHECK

// MEDIUM

Description

The `GetCurrentHeight` method in the `P2PValidatorClient` class contains a logical inconsistency between the connectivity validation and height aggregation phases. The method correctly validates validator connectivity using `AreValidatorsConnected()` and updates validator node heights through `UpdateNodeHeights()`, which operates exclusively on `Globals.ValidatorNodes`. However, the subsequent height calculation loop iterates over `Globals.Nodes.Values` instead of `Globals.ValidatorNodes.Values`, creating a mismatch between the validated peer set and the aggregation source.

This inconsistency can lead to stale or incorrect height decisions because the method may consider heights from general peers that were not updated in the validator-specific `UpdateNodeHeights()` call. Since validator nodes and general nodes can have different connectivity states and update frequencies, this mismatch could result in synchronization decisions based on outdated information from non-validator peers while ignoring fresh data from the validated validator peers.

The method gates execution on validator connectivity and updates validator heights, but then aggregates heights from a completely different peer collection, potentially causing the validator to make synchronization decisions based on stale data from general peers instead of current validator consensus.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

1. Align the aggregation set with the gating/updates: Iterate `Globals.ValidatorNodes.Values` for height aggregation (same set as `AreValidatorsConnected()` and `UpdateNodeHeights()`).
2. If general peers must be considered, explicitly refresh their heights in the same path and gate with a separate, documented policy; weigh validator heights preferentially.
3. Add defensive checks (connected, timestamp freshness) before using a peer's `NodeHeight`.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/c22923d832b854f85fe9f80075b4e537d4c0e3e7

7.17 AUTHENTICATION HANDSHAKE VULNERABLE TO REPLAY ATTACKS

// MEDIUM

Description

The validator and blockcaster authentication handshakes in `P2PValidatorServer` and `P2PBlockcasterServer` accept timestamps within a 300-second window and lack replay protection mechanisms. The handshake process validates signatures using a message format of `address + ":" + time + ":" + publicKey`, but only checks that the timestamp is within 5 minutes of the current time using `TimeUtil.GetTime() - long.Parse(time) > 300`. This creates a significant replay attack window where malicious actors who observe legitimate authentication attempts can reuse the same signed credentials to gain unauthorized access to validator or blockcaster endpoints.

The vulnerability stems from the absence of any nonce or one-time challenge mechanism that would bind each authentication attempt to a unique session. Since the signed message format remains constant for a given address and public key combination within the 300-second window, captured authentication credentials remain valid for replay attacks throughout this entire period. Additionally, the use of `long.Parse(time)` without validation could potentially cause denial of service through malformed timestamp values that trigger parsing exceptions.

Both server implementations share this identical flawed authentication pattern, making the vulnerability systemic across the validator network infrastructure. Successful replay attacks could enable unauthorized retention of access to consensus channels, facilitating Sybil attacks or other malicious behaviors within the validator network.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:M/A:N/I:N/D:N/Y:N (5.0)

Recommendation

- Reduce window to ≤30 seconds with bounded clock skew.
- Add per-attempt nonce (unique, TTL) and reject re-use.
- Bind signed content to stable peer identity (address ↔ public key mapping) and validate consistency.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/ee3471ff6d9a33cfe181d5cd7934bb21737fce94

7.18 UNCAUGHT PARSING EXCEPTION ENABLES HANDSHAKE DENIAL OF SERVICE

// MEDIUM

Description

The handshake authentication logic in both `P2PBlockcasterServer` and `P2PValidatorServer` uses `long.Parse(time)` to process the timestamp header provided by connecting clients without proper input validation. This creates a denial of service vulnerability where malicious actors can send malformed timestamp values that trigger parsing exceptions, causing the connection handler to abort unexpectedly. The vulnerable code directly parses user-controlled input using `TimeUtil.GetTime() - long.Parse(time) > 300` without validating that the input is a properly formatted numeric value.

When an attacker sends non-numeric timestamp values such as alphabetic strings, special characters, or numeric values that exceed the valid range for `long` data types, the `long.Parse` method throws format or overflow exceptions. These unhandled exceptions can disrupt the authentication process and potentially impact the availability of validator and blockcaster endpoints under sustained attack scenarios. The lack of input validation allows remote attackers to repeatedly trigger these exceptions without requiring any authentication or proof of stake.

Both server implementations share this identical vulnerability pattern, making the issue systemic across the validator network infrastructure. The vulnerability affects the critical handshake process that determines whether nodes can join validator consensus channels, potentially degrading network availability during coordinated attacks.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

- Use safe parsing (`TryParse`) and reject on failure.
- Enforce strict bounds on timestamp skew and type (Unix seconds only).
- Fail closed with minimal error disclosure; consider rate-limiting/temporary ban on repeated invalid inputs.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/9403c3434ca4b2da0ad4dcd0c1670d255137a9aa

7.19 UNAUTHENTICATED VALIDATOR DISCOVERY ENABLES NETWORK TOPOLOGY MANIPULATION

// MEDIUM

Description

The `P2PValidatorClient` class implements a validator discovery mechanism in the `RequestActiveValidators` method that accepts and persists validator entries from any connected validator peer without implementing adequate safeguards against malicious injection. The method invokes `SendActiveVals` on connected validator peers and processes the decompressed JSON response containing lists of `NetworkValidator` objects, immediately adding valid entries to both the in-memory `Globals.NetworkValidators` collection and the persistent peer database.

While the `NetworkValidator.AddValidatorToPool` method performs individual signature verification on each validator record using `SignatureService.VerifySignature(validator.Address, validator.SignatureMessage, validator.Signature)` , the discovery process lacks critical protections including quorum validation, cross-verification between multiple independent sources, and rate limiting per advertising peer.

This design allows a single malicious or compromised validator to flood the local node's validator registry with arbitrary entries, potentially including validators under the attacker's control or non-existent validators that could bias peer selection algorithms and network routing decisions. The vulnerability is particularly concerning because validator discovery directly affects consensus participation, block propagation paths, and the overall security of the distributed validator network.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Require cryptographic attestation per validator entry: Each advertised validator should sign a registry payload (address, publicKey, IP, timestamp/nonce). Verify address and publicKey binding. Optionally, require adjudicator or quorum co-signature over snapshots.
- Cross-check multiple independent sources before accepting/persisting; reject entries not corroborated.
- Enforce per-source rate limits and caps; avoid persisting entries that fail attestation or freshness checks.
- Prefer pulling from a consensus-controlled registry (on-chain or coordinator-signed) rather than trusting single peer pushes.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/fe79406aec7e7ff0e0d8f2744ef13dd1e9ce5592

7.20 UNSAFE STRING SLICING ON UNTRUSTED WALLET VERSION CAUSES CONNECTION FAILURES

// MEDIUM

Description

The `P2PValidatorClient` class contains a robustness vulnerability in both the `Connect` method (around line 227) and `ConnectBlockcaster` method (around line 513) where it processes wallet version strings received from remote validator peers without adequate length validation. After invoking `GetWalletVersion(node.Connection)` which calls `hubConnection.InvokeAsync<string>("GetWalletVersion")` on the remote peer, the code only performs a null check before directly calling `walletVersion.Substring(0,3)` to extract the first three characters.

This implementation assumes that any non-null wallet version string will have at least 3 characters, but malicious or misconfigured peers can return empty strings, single characters, or two-character strings that cause `ArgumentOutOfRangeException` when the substring operation attempts to access indices beyond the string length. The vulnerability affects both validator connection establishment and block caster connection processes.

The transport layer's disabled certificate validation (`RemoteCertificateValidationCallback = (sender, cert, chain, errors) => true`) compounds this issue by enabling man-in-the-middle attackers to intercept and modify wallet version responses in transit, allowing external attackers to trigger the same parsing failures that malicious peers could cause directly.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

Implement proper input validation before string slicing by checking both null status and minimum length requirements using conditions like `!string.IsNullOrEmpty(walletVersion) && walletVersion.Length >= 3` before calling `Substring(0,3)` .

Consider implementing more robust version parsing using semantic version validation or regex patterns that can properly validate version string formats and reject malformed inputs gracefully.

Add proper exception handling around the substring operations so that invalid wallet version strings cause the peer connection to be rejected cleanly without throwing unhandled exceptions that could disrupt the connection establishment process.

Implement fallback behavior for peers with invalid wallet versions by treating them as non-compliant and either assigning a default version or skipping the connection entirely while logging the issue at an appropriate level to avoid log spam during potential attack scenarios.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/533df68531844b8fa17f0544fa44283b0901e872

7.21 ADDRESS-BASED BANS APPLIED BEFORE AUTHENTICATION

// MEDIUM

Description

The contract `P2PValidatorServer` performs an address blocklist (ABL) check in the `OnConnectedAsync` function by comparing the unauthenticated `address` header against the blocklist and banning the peer's IP if a match is found. This check occurs before verifying the caller's signature with `VerifySignature(address, SignedMessage, signature)`, meaning the `address` value is not yet authenticated and can be spoofed by an attacker. As a result, a malicious peer can supply any address in the connection header to trigger bans on arbitrary IPs or evade ABL checks by presenting a benign address initially and switching to a different authenticated identity after passing the check. This weakens the trust and effectiveness of the ban mechanism and can be abused for targeted denial-of-service or policy bypass.

Proof of Concept

```
var ablList = Globals.ABL.ToList();

if (ablList.Exists(x => x == address))
{
    BanService.BanPeer(peerIP, "Request malformed", "OnConnectedAsync");
    await EndOnConnect(peerIP, $"ABL Detected", $"ABL Detected: {peerIP}.");
    return;
}
// ... signature verification happens after this check ...
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Move ABL enforcement after successful signature verification, binding the ban decision to the authenticated `address` (and optionally `publicKey`).
- Prefer identity-based bans (authenticated validator) over IP-only bans; combine with short-lived IP throttling if necessary.
- Add bounded logging and rate limits for ban actions, and consider requiring repeated offenses before applying long-term bans.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/bcd88da38f976051db477954902a420ece3682f5

7.22 UNBOUNDED FIELD LENGTHS IN HANDSHAKE AND VALIDATOR MODELS

// MEDIUM

Description

The contract `P2PValidatorServer` accepts and stores handshake headers such as `uName` and `publicKey`, as well as `NetworkValidator` model fields, without enforcing explicit maximum lengths or strict schema validation. These fields are later stored and broadcast to other peers, including through the `SendNetworkValidatorList` function, without any checks on their size or content. As a result, an attacker can supply oversized or malformed values for these fields, leading to inflated memory and serialization costs, increased outbound payload sizes, and potential bandwidth exhaustion. This lack of validation can also cause downstream failures in consumers that expect bounded field sizes.

Proof of Concept

- Handshake headers (`uName`, `publicKey`) are read and stored without length checks.
- `SendNetworkValidatorList` accepts and broadcasts `NetworkValidator` objects with unbounded string fields.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Define and enforce maximum lengths and allowed character sets for `uName`, `publicKey`, and all `NetworkValidator` fields.
- Validate models on receipt, rejecting or truncating (with logging) any inputs that exceed defined limits.
- Ensure that all broadcasts only include bounded, sanitized fields to prevent propagation of oversized or malformed data.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/1a2e584b686fab4ff3ff16697330398d052031a7

7.23 FIRE-AND-FORGET BROADCASTS WITHOUT ERROR HANDLING OR FLOW CONTROL

// MEDIUM

Description

The contract `P2PValidatorServer` issues broadcast calls using fire-and-forget semantics, such as `_ = Clients.Caller.SendAsync(...)` and `_ = Clients.All.SendAsync(...)`, without awaiting the returned tasks or implementing any error handling or backpressure mechanisms. As a result, exceptions that occur during the send operations are never observed, failed sends are silent, and the server cannot apply retries or flow control. This approach degrades the reliability of network communication, can mask underlying network problems, and allows bursts of broadcasts to accumulate unbounded work, contending for CPU and network resources.

Proof of Concept

```
_ = Clients.Caller.SendAsync("GetValMessage", "1", peerIP, new CancellationTokenSource(2000).Token);
_ = Clients.All.SendAsync("GetValMessage", "3", netValSerialize, new CancellationTokenSource(6000).Token);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Await all `SendAsync` calls with bounded cancellation and error handling using `try/catch`; log failures at a bounded rate and apply retries if needed.
- If asynchronous dispatch is required, enqueue sends to a bounded producer/consumer queue, process with limited concurrency, and capture task failures for metrics and logs.
- Rate-limit and deduplicate join broadcasts by caching the last broadcast per address and suppressing repeats within a short TTL; batch multiple joins into a single periodic update.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/2c277f1e2ddf9c024f0e43af00f7f70b862f7692

7.24 MISSING VALIDATOR ROLE ASSERTION IN BLOCK RECEPTION

// MEDIUM

Description

The contract `P2PValidatorServer` allows block submissions through the `ReceiveBlockVal` function without explicitly asserting that the caller is an authenticated validator authorized to submit blocks. The method relies on the existing Hub connection but does not check whether the caller's context is flagged as a validator after passing stake or balance checks. This omission allows unauthorized or unauthenticated callers to trigger block processing logic, increasing the risk of unsolicited block push attempts, unnecessary processing, and easier probing or fuzzing of the endpoint. The blast radius is further increased if a connection is compromised, as there is no role-based restriction on who can invoke this sensitive operation.

Proof of Concept

- `ReceiveBlockVal` processes incoming blocks without checking if the caller is a validated and authorized validator.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- On connection, mark the Hub context with a validator role after successful stake and balance verification.
- In `ReceiveBlockVal`, assert the validator role and reject requests from unauthorized callers.
- Log rejected attempts at a bounded rate to monitor abuse without overwhelming logs.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/6f2ad36fc61c9636356bc03c053a428f2b9888af

7.25 SYNCHRONOUS REMOTE PORT CHECK IN HANDSHAKE PATH

// MEDIUM

Description

The contract `P2PValidatorServer` performs a synchronous remote port check on the peer's IP address during the `OnConnectedAsync` handshake, using `PortUtility.IsPortOpen(peerIP, Globals.ValPort)` before proceeding with authentication. This check adds latency to the handshake process and can be abused by attackers to force repeated port probes, consuming server CPU and socket resources. The port check is also an unreliable admission signal, as network conditions (NAT/firewall state) may change between the check and actual communication, and successful SignalR connectivity already demonstrates reachability. Admission decisions are thus tied to transient network conditions rather than authenticated protocol or identity checks.

Proof of Concept

```
var portCheck = PortUtility.IsPortOpen(peerIP, Globals.ValPort);
if(!portCheck)
{
    _ = EndOnConnect(peerIP, $"Port: {Globals.ValPort} was not detected as open.", $"Port: {Globals.ValPort} was n
    return;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

- Remove the port probe from the handshake path; rely on successful Hub connection and authenticated identity (signature and stake checks) for admission.
- If port probing is needed for diagnostics, run it asynchronously off the handshake path, with tight timeouts and cached results (TTL), not as a connection gate.
- Base admission policy on protocol and authentication criteria instead of ad-hoc remote port scans.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/cfa8fb588cd3bd417b2a2a4ca168a4e3fe09cc6c

7.26 MISSING ADDRESS–PUBLICKEY BINDING IN VALIDATOR HANDSHAKE

// MEDIUM

Description

The contract `P2PValidatorServer` verifies a signature over the concatenated string `address:time:publicKey` during the validator handshake, but does not explicitly check that the supplied `publicKey` actually derives the claimed `address` according to the chain's address derivation rules. The server then stores the unverified `publicKey` alongside the `address` in the `NetworkValidator` record. This creates ambiguity in identity binding and allows inconsistent or malicious pairings of addresses and public keys to be admitted and persisted. If downstream logic relies on the stored `PublicKey`, this can lead to mismatches, weaken the trust model for validator metadata, and complicate subsequent authorization or registry updates.

Proof of Concept

```
SignedMessage = address + ":" + time + ":" + publicKey;
var verifySig = SignatureService.VerifySignature(address, SignedMessage, signature);
// No check that publicKey derives address
var netVal = new NetworkValidator {
    Address = address,
    IPAddress = peerIP.Replace("::ffff:", ""),
    PublicKey = publicKey,
    Signature = signature,
    SignatureMessage = SignedMessage,
    UniqueName = uName,
};
Globals.NetworkValidators.TryAdd(address, netVal);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Enforce address–publicKey binding at handshake by deriving the expected address from the provided `publicKey` and requiring it matches the claimed `address`.
- Alternatively, verify the signature using the provided `publicKey`, then independently confirm that `publicKey` corresponds to `address`.
- Validate the format and length of `publicKey` before use and reject malformed keys.
- Only persist `PublicKey` to `Globals.NetworkValidators` after the binding check passes.
- Remove weak checks elsewhere (such as `Signature.Contains(publicKey)`) and rely on proper cryptographic verification over a canonical payload including address, publicKey, IP, and timestamp/nonce.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/61c7b0e093a7e4e1e834e0d4818df6f748dced66

7.27 SILENT EXCEPTION HANDLING HIDES ABUSE AND OPERATIONAL FAULTS

// MEDIUM

Description

The contract `P2PValidatorServer` contains multiple handler paths where exceptions are caught and ignored, either with empty `catch { }` blocks or by catching exceptions and not logging or acting on them. This pattern appears in methods such as `SendNetworkValidatorList`, `ReceiveBlockVal`, `SendQueuedBlock`, `SendWinningProofVote`, and `SendTxToMempoolVals`. Silent failures hide abuse and operational faults, hinder incident response and triage, and allow repeated exception triggers without visibility. As a result, malformed or oversized payloads can repeatedly cause exceptions without alerting operators, and partial failures (such as broadcast or queue insert errors) can degrade liveness and reliability without detection.

Proof of Concept

```
public async Task SendNetworkValidatorList(string data)
{
    try { ... }
    catch { }
}

public async Task<bool> ReceiveBlockVal(Block nextBlock)
{
    try { ... }
    catch { }
    return false;
}

public async Task<Block?> SendQueuedBlock(long currentBlock)
{
    try { ... }
    catch { }
    return null;
}

public async Task SendWinningProofVote(string winningProofJson)
{
    try { ... }
    catch (Exception ex)
    {
    }
}

public async Task<string> SendTxToMempoolVals(Transaction txReceived)
{
    try { return await SignalRQueue(...); }
    catch { }
    return "TFVP";
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Replace empty catch blocks with bounded logging and metrics: log the peer IP, method name, and a compact error code; rate-limit logs to prevent flooding and export counters for monitoring.
- Return explicit failure codes or paths, surfacing validation errors cleanly and applying cooldowns or bans on repeated faults per IP.
- Narrow the scope of exception handling to the smallest block that can handle recovery, avoiding blanket try/catch around entire methods.
- Wrap async sends with error handling: always `await` with cancellation and `try/catch`, and handle faults explicitly (never use fire-and-forget without observation).

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/85234d232d43e9aa73becd607047e6476ee42ace

7.28 UNBOUNDED WINNER LIST RESPONSES WITHOUT PAGINATION OR SIZE LIMITS

// MEDIUM

Description

The contract `P2PValidatorServer` exposes endpoints `GetWinningProofList` and `GetFinalizedWinnersList` that can return arbitrarily large lists of winning proofs or finalized winners without any pagination, serialized size limits, or field trimming. These methods serialize and return the entire list of entries matching the query, regardless of the total size. Under conditions of large state or request floods, these responses can cause significant CPU and bandwidth spikes, and repeated requests can amplify the impact. Additionally, these endpoints return network state to callers without authentication or gating, increasing the risk of information exposure.

Proof of Concept

```
public async Task<string> GetWinningProofList()
{
    string result = "0";
    if(Globals.WinningProofs.Count() != 0)
    {
        var heightMax = Globals.LastBlock.Height + 10;
        var list = Globals.WinningProofs.Where(x => x.Key <= heightMax).Select(x => x.Value).ToList();
        if(list != null)
            result = JsonConvert.SerializeObject(list);
    }
    return result;
}

public async Task<string> GetFinalizedWinnersList()
{
    string result = "0";
    if (Globals.WinningProofs.Count() != 0)
    {
        var list = Globals.FinalizedWinner.Select(x => x.Value).ToList();
        if (list != null)
            result = JsonConvert.SerializeObject(list);
    }
    return result;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Paginate or window results and enforce a hard serialized byte cap per response.
- Gate access to these endpoints by requiring an authenticated validator role and apply per-IP rate limits.
- Cache trimmed snapshots with a TTL to avoid heavy per-call serialization.
- Return only necessary fields and avoid including sensitive metadata in responses.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.29 BLOCK BROADCAST NOT GATED ON VALIDATION SUCCESS

// MEDIUM

Description

The contract `P2PValidatorServer` may broadcast a received block to all peers in the `ReceiveBlockVal` method without ensuring that the block has been successfully validated. While validation is invoked via `BlockValidatorService.ValidateBlocks()`, the subsequent broadcast is not conditioned on the result of this validation. This can lead to the propagation of unverified or invalid blocks, causing peers to spend resources parsing and rejecting them, and amplifying network and CPU load, especially under spam or flood conditions.

Proof of Concept

```
if(Globals.LastBlock.Height < nextBlock.Height)
    await BlockValidatorService.ValidateBlocks();

if (nextHeight == currentHeight)
{
    string data = "";
    data = JsonConvert.SerializeObject(nextBlock);
    await Clients.All.SendAsync("GetMessage", "blk", data);
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Gate block broadcasts on explicit success from `BlockValidatorService` by checking the per-block validation result or a cache flag before broadcasting.
- Maintain a set of validated blocks and only broadcast entries that are present in this set.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/49a5635cecbca878bc5b11b8075b2f0c649547ab

7.30 EXCESSIVE PARALLEL REQUESTS AND STATIC BACKOFF

// MEDIUM

Description

In the consensus messaging logic of the affected contract, the use of `ParallelLoop` to send concurrent Message and Hash requests to all peers with short delays and fixed heartbeat timeouts can result in continuous retries and excessive logging when partial failures occur. This behavior increases CPU and network load, amplifies logs, and can lead to self-induced denial-of-service, especially if packet loss or slow peers are present. The issue is present in the consensus messaging implementation, where requests are sent without adaptive throttling or per-peer failure management.

Proof of Concept

The following code from the consensus messaging contract demonstrates the issue:

```
peers.ParallelLoop(peer =>
{
    _ = MessageRequest(peer, ToSend, currentHeight, methodCode, messages, addresses, cts);
});
...
await cts.Token.WhenCanceled();
```

```
peers.ParallelLoop(peer =>
{
    _ = HashRequest(peer, ToSend, currentHeight, methodCode, hashes, addresses, cts);
});
```

When some peers are slow or unresponsive, these loops continuously retry requests, causing increased resource consumption and log amplification.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Apply bounded concurrency and adaptive backoff strategies, such as exponential or jittered delays, to limit resource usage.
- Track a per-peer failure budget and pause or deprioritize noisy peers to prevent resource exhaustion.
- Batch or coalesce requests when many peers are outstanding to reduce network and processing overhead.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f71fee70db9e2b38a396fedc7d5b7808e826bd95

7.31 UNSAFE RESPONSE PARSING RISKS EXCEPTIONS AND DESYNCHRONIZATION

// MEDIUM

Description

In the consensus contract, client code splits untrusted response strings and directly parses array indices and numeric fields using `int.Parse` and `long.Parse` without validating array lengths or using safe parsing methods. Malformed or crafted responses can trigger `IndexOutOfRangeException` or `FormatException`, disrupting consensus loops and potentially desynchronizing state. This exposes the contract to availability risks, as any reachable peer can send malformed frames to repeatedly trigger error paths and slow consensus progress.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
var PrefixSplit = Response.Split(new[] { '|' }, 2);
var Prefix = PrefixSplit[0].Split(':');
...
peer.NodeHeight = long.Parse(Prefix[0]);
peer.MethodCode = int.Parse(Prefix[1]);
peer.IsFinalized = Prefix[2] == "1";
...
var arr = PrefixSplit[1].Split("::;");
var (address, message, signature) = (arr[0], arr[1].Replace("::", ":"), arr[2]);
```

Malformed responses such as `|` or `1:abc|` can cause out-of-range indexing or parsing errors, leading to exceptions and consensus disruption.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Validate array lengths before indexing and use `TryParse` with guard clauses to prevent exceptions from malformed input.
- Enforce maximum sizes for message and hash fields, rejecting oversized frames to avoid resource exhaustion.
- Treat unknown or malformed formats as benign no-ops, logging metrics for monitoring rather than mutating contract state.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.32 UNCHECKED SPLIT-BASED PARSING IN MESSAGE/HASH ENDPOINTS

// MEDIUM

Description

In the consensus contract, the `Message` and `Hash` endpoints parse untrusted input using `Split` and positional indexing without validating array lengths or input sizes. Malformed frames can trigger exceptions or be dropped silently, while large inputs may cause excessive memory usage and log amplification. This lack of bounds and size checks degrades consensus throughput and exposes the contract to memory and log-based denial-of-service risks.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
if(peerMessage != null)
{
    var split = peerMessage.Split("::;");
    (message, signature) = (split[0], split[1]);
    message = message.Replace("::", ":");
}
```

```
if (peerHash != null)
{
    var split = peerHash.Split(":");
    (hash, signature) = (split[0], split[1]);
}
```

Malformed or oversized frames can cause exceptions or excessive resource consumption.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Validate array length before indexing and enforce a maximum frame size; use guarded parsing similar to `TryParse`.
- Treat invalid frames as benign no-ops, recording metrics and applying optional rate-limits, while avoiding verbose logging on bad inputs.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f70ab47b2bc63cddf8778e1a115cf248031e6ef8

7.33 UNBOUNDED TRANSACTION BROADCAST LIST INGESTION

// MEDIUM

Description

In the consensus contract, the server accepts `List<TransactionBroadcast>` without explicit caps or admission control. This allows a malicious peer to submit extremely large lists, triggering excessive serialization, hashing, and dictionary insertions. Such behavior can lead to spikes in CPU and memory usage, risking out-of-memory conditions and performance degradation. The lack of request size limits and rate controls makes the contract vulnerable to resource exhaustion.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
public async Task<bool> GetBroadcastedTx(List<TransactionBroadcast> txBroadcastList)
{
    bool result = false;
    try
    {
        var txHashes = JsonConvert.SerializeObject(txBroadcastList.Select(x => x.Hash).ToArray());
        AdjLogUtility.Log($"UTC Time: {DateTime.Now} TX List Received: {txHashes}", "ConsensusServer.GetBroadcaste
        foreach (var txBroadcast in txBroadcastList)
        {
            if (!Globals.ConsensusBroadcastedTrxDict.TryGetValue(txBroadcast.Hash, out _))
            {
                var isTxStale = await TransactionData.IsTxTimestampStale(txBroadcast.Transaction);
                if(!isTxStale)
                {
                    var isCraftedIntoBlock = await TransactionData.HasTxBeenCraftedIntoBlock(txBroadcast.Transacti
                    if (!isCraftedIntoBlock)
                    {
                        Globals.ConsensusBroadcastedTrxDict[txBroadcast.Hash] = new TransactionBroadcast
                        {
                            Hash = txBroadcast.Hash,
                            IsBroadcastedToAdj = false,
                            IsBroadcastedToVal = false,
                            Transaction = txBroadcast.Transaction,
                            RebroadcastCount = 0
                        };
                    }
                    else
                    {
                        Globals.BroadcastedTrxDict.TryRemove(txBroadcast.Hash, out _);
                        Globals.ConsensusBroadcastedTrxDict.TryRemove(txBroadcast.Hash, out _);
                    }
                }
                else
                {
                    Globals.BroadcastedTrxDict.TryRemove(txBroadcast.Hash, out _);
                    Globals.ConsensusBroadcastedTrxDict.TryRemove(txBroadcast.Hash, out _);
                }
            }
        }
        result = true;
    }
    catch(Exception ex)
    {
        result = false;
        AdjLogUtility.Log($"Error receiving broadcasted TX. Error: {ex.ToString()}", "ConsensusServer.GetBroadcast
        ErrorLogUtility.LogError($"Error receiving broadcasted TX. Error: {ex.ToString()}", "ConsensusServer.GetBr
```

```
    }
    return result;
}
```

A peer can submit a very large `txBroadcastList`, causing excessive resource consumption.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

- Enforce a maximum list length and total byte size per request; apply per-peer rate limits to prevent resource exhaustion.
- Short-circuit early by validating headers or hashes only, and avoid serializing entire lists in logs.
- Consider background admission with bounded queues to control processing load.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/46edb711d67c7c5435650766ce08ae6d242de31f

7.34 UNSAFE TIMESTAMP PARSING AND MISSING NONCE IN HANDSHAKE

// MEDIUM

Description

In the consensus contract, `OnConnectedAsync` parses the `time` header using `long.Parse` without validating its presence or format, and applies only a 15-second window for freshness. This exposes the contract to exception-driven denial-of-service if malformed values are received, and allows replay attacks within the time window due to the absence of a nonce in the signed envelope.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
var time = httpContext.Request.Headers["time"].ToString();
...
var Now = TimeUtil.GetTime();
if (Math.Abs(Now - long.Parse(time)) > 15)
{
    EndOnConnect(peerIP, "Signature Bad time.", "Signature Bad time.");
    return;
}
```

Malformed or missing `time` values can trigger `FormatException`, and lack of nonce enables replay within the allowed window.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

- Use `long.TryParse` to safely parse the timestamp and reject invalid or overflow values with clear logging.
- Include a per-connection nonce in the signed message, such as signing `address:time:nonce`, and track nonce uniqueness to prevent replay.
- Consider making the time window configurable and add tolerance for future-skew.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.35 INVERTED DUPLICATE HANDLING LOGIC IN TASK ANSWER PROCESSING

// MEDIUM

Description

The `ReceiveTaskAnswerV3` function contains inverted logic when handling duplicate task answer submissions to `Globals.TaskAnswerDictV3`. When `TryAdd` fails (indicating a duplicate entry already exists), the method incorrectly returns `AnswerAccepted = true` with `AnswerCode = 0`, treating the duplicate as a successful submission. Conversely, when `TryAdd` succeeds (first valid submission), the method sets `AnswerCode = 7` with the comment "Answer was already submitted", rejecting the legitimate first answer. This logic inversion allows duplicate submissions to be accepted while rejecting valid initial submissions, potentially corrupting consensus mechanisms and answer validation processes.

Proof of Concept

```
var taskResult = request?.Split(':');
...
var (Answer, Height) = (int.Parse(taskResult[0]), long.Parse(taskResult[1]));
...
if (!Globals.TaskAnswerDictV3.TryAdd((Pool.Key2, Height), (ipAddress, Pool.Key2, Answer)))
{
    taskAnsRes.AnswerAccepted = true;
    taskAnsRes.AnswerCode = 0;
    return taskAnsRes;
}

taskAnsRes.AnswerCode = 7; // Answer was already submitted
return taskAnsRes;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Correct the logic flow so that successful `TryAdd` operations set `AnswerAccepted = true` and `AnswerCode = 0` to properly accept the first valid submission. • Handle `TryAdd` failure cases by setting `AnswerAccepted = false` and appropriate non-zero error codes such as `AnswerCode = 7` to reject duplicate submissions. • Implement comprehensive idempotency validation checks and create unit tests to verify correct duplicate detection and acceptance logic behavior across various submission scenarios.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/667066a172907cceaee5a2c7bde1dac1a972a62d

7.36 IP-ONLY GATING ENABLES SESSION HIJACK AND MISROUTING

// MEDIUM

Description

Multiple beacon message routing flows rely on weak reference-based lookups using `beaconData.Reference` or `beaconData.NextOwnerReference` without cryptographic authentication binding. The system routes messages to recipients by matching reference values in the beacon pool and sending data to the corresponding `ConnectionId`. An attacker sharing the same IP address or NAT, or capable of predicting or spoofing reference values, can hijack asset delivery signals and misdirect messages intended for legitimate users. This lack of strong authentication binding enables session hijacking and privacy leakage through message interception.

Proof of Concept

```
var receiverRef = beaconData.NextOwnerReference;
var remoteUser = beaconPool.Where(x => x.Reference == receiverRef).FirstOrDefault();
if (remoteUser != null)
{
    string[] senddata = { beaconData.SmartContractUID, beaconData.AssetName };
    var sendJson = JsonConvert.SerializeObject(senddata);
    await SendMessageClient(remoteUser.ConnectionId, "receive", sendJson);
}
```

```
if (Globals.BeaconPool.TryGetFromKey2(beaconData.Reference, out var remoteUser))
{
    string[] senddata = { beaconData.SmartContractUID, beaconData.AssetName };
    var sendJson = JsonConvert.SerializeObject(senddata);
    if (remoteUser.Value != null)
    {
        await SendMessageClient(remoteUser.Value.ConnectionId, "send", sendJson);
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Implement cryptographic authentication binding by associating reference values with authenticated public keys or verified user identities before allowing message routing. • Include signed nonces in message routing protocols and verify ownership through cryptographic signatures before forwarding messages to recipients. • Add short expiration times to reference mappings in the beacon pool and implement periodic cleanup to minimize the window of opportunity for hijacking attacks.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/2985f087f3e1d7c00af2fe1070c34c01936f0f3a

7.37 UNSAFE TIMESTAMP PARSING AND MISSING NONCE IN BLOCKCASTER HANDSHAKE

// MEDIUM

Description

The `OnConnectedAsync` method implements unsafe timestamp validation that creates multiple security vulnerabilities. The function uses `long.Parse(time)` to process user-supplied timestamp data without validation, making it susceptible to format exceptions when malformed input is provided. The signed message format `address + ":" + time + ":" + publicKey` lacks nonce inclusion, enabling replay attacks within the 300-second acceptance window. No signature reuse validation is implemented at this layer, allowing attackers to intercept and replay valid authentication messages multiple times before expiration, potentially bypassing intended access controls.

Proof of Concept

```
var SignedMessage = address;
var Now = TimeUtil.GetTime();
SignedMessage = address + ":" + time + ":" + publicKey;
if (TimeUtil.GetTime() - long.Parse(time) > 300)
{
    _ = EndOnConnect(peerIP, "Signature Bad time.", "Signature Bad time.");
    return;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Replace unsafe parsing with `long.TryParse` and implement explicit range validation to handle malformed timestamps gracefully and prevent format exceptions from causing service disruption.
• Enhance the signature scheme by including a per-connection nonce in the signed message format, changing to `address:time:publicKey:nonce` , and implement nonce uniqueness tracking with appropriate time-to-live mechanisms.
• Reduce the time acceptance window from 300 seconds and add configurable future-skew tolerance to minimize replay attack opportunities while handling legitimate clock synchronization differences.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.38 RECEIVEBLOCKVAL HEAVY PATH WITHOUT SIGNALRQUEUE/DOS GUARD

// MEDIUM

Description

The `ReceiveBlockVal` method processes incoming block submissions without implementing proper throttling mechanisms, creating a resource exhaustion vulnerability. The `SignalRQueue` rate limiting functionality has been commented out, allowing unthrottled submission of large blocks that trigger expensive validation operations. Each request initiates multiple resource-intensive operations including block validation via `BlockValidatorService.ValidateBlocks()`, JSON serialization through `JsonConvert.SerializeObject(nextBlock)`, and broadcast operations to all connected clients via `Clients.All.SendAsync`. The absence of size limits and per-IP rate controls enables attackers to overwhelm the system with computationally expensive block processing requests.

Proof of Concept

```
public async Task<bool> ReceiveBlockVal(Block nextBlock)
{
    try
    {
        //return await SignalRQueue(Context, (int)nextBlock.Size, async () =>
        //{
        if (nextBlock.ChainRefId == BlockchainData.ChainRef)
        {
            var IP = GetIP(Context);
            var nextHeight = Globals.LastBlock.Height + 1;
            var currentHeight = nextBlock.Height;
            if (currentHeight >= nextHeight && BlockDownloadService.BlockDict.TryAdd(currentHeight, (nextBlock, IP
            {
                await Task.Delay(2000);
                if (Globals.LastBlock.Height < nextBlock.Height)
                    await BlockValidatorService.ValidateBlocks();
                if (nextHeight == currentHeight)
                {
                    string data = "";
                    data = JsonConvert.SerializeObject(nextBlock);
                    await Clients.All.SendAsync("GetCasterMessage", "blk", data);
                }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Re-enable the `SignalRQueue(Context, (int)nextBlock.Size, ...)` mechanism with appropriate quotas to enforce rate limiting and prevent resource exhaustion from excessive block submissions.
• Implement maximum block size validation and per-IP rate limiting controls to restrict the volume and frequency of block processing requests from individual peers.

• Add early validation checks to verify lightweight block headers and chain references before queuing computationally expensive validation and broadcast operations, reducing unnecessary resource consumption for invalid blocks.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/156b8c1984a7afcedebd0b9baaa94c50fdf9594e

7.39 AGGRESSIVE PARALLELISM/BACKOFF IN PEER CONNECTIONS AND UPDATES

// MEDIUM

Description

The P2P client implementation launches unbounded parallel operations for peer connections and method code updates without implementing central backpressure controls. The `ParallelLoop` operations on `newPeers.Take(Diff).ToArray()` and `Globals.Nodes.Values.ToArray()` create concurrent connection attempts and update requests that can overwhelm system resources. Under adverse network conditions or during connection churn, these parallel operations stack retries and timeouts without coordination, leading to CPU and thread pool pressure, queue contention, and increased connection drop rates that can cascade into further instability.

Proof of Concept

```
newPeers.Take(Diff).ToArray().ParallelLoop(peer =>
{
    _ = Connect(peer);
});

Globals.Nodes.Values.ToArray().ParallelLoop(node =>
{
    if (node.Address != Address && !UpdateMethodCodeAddresses.ContainsKey(node.NodeIP))
    {
        UpdateMethodCodeAddresses[node.NodeIP] = true;
        _ = UpdateMethodCode(node);
    }
});
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Implement bounded task schedulers with configurable concurrency limits to control the maximum number of simultaneous connection attempts and update operations.
• Add staggered retry mechanisms with exponential backoff to prevent thundering herd effects and reduce system load during network instability.
• Establish centralized backpressure controls and per-IP quotas to manage resource consumption and prevent individual peers from overwhelming the connection pool through excessive parallel operations.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/bfdf82b2724b60646dc20b91082de5a4afc3dd18

7.40 IP-KEYED SESSION MAPPING ENABLES HIJACK/MISDIRECTION

// MEDIUM

Description

The session management system relies solely on IP addresses as unique identifiers for connection mapping, creating vulnerabilities in shared network environments. The `Globals.MothersKidsContext` mapping uses `peerIP` as the primary key without additional authentication binding, allowing attackers behind the same NAT or those capable of IP spoofing to interfere with legitimate sessions. When a new connection is detected from the same IP, the system aborts the existing connection via `context.Abort()` and replaces the session mapping, enabling session hijacking and misdirection attacks that can disrupt legitimate peer communications.

Proof of Concept

```
var peerIP = GetIP(Context);
if (Globals.MothersKidsContext.TryGetValue(peerIP, out var context) && context.ConnectionId != Context.ConnectionI
    context.Abort();

Globals.MothersKidsContext[peerIP] = Context;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Replace IP-based session identification with cryptographically authenticated identity binding using public key authentication or signed session tokens.
• Implement per-connection nonce mechanisms and unique connection identifiers that cannot be easily guessed or replicated by attackers sharing network infrastructure.
• Add session validation checks that verify ownership through cryptographic proofs before allowing session replacement or modification operations.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/06e138b6d951daabf00b9be599d957306c9f276c

7.41 DOS THROTTLING WEAKNESSES IN SIGNALRQUEUE

// MEDIUM

Description

The `SignalRQueue` throttling mechanism implements per-IP rate limiting with simplistic thresholds that can be circumvented by coordinated attacks. The system uses connection count limits (`Lock.ConnectionCount > 20`) and buffer cost tracking (`Lock.BufferCost + sizeCost > 5000000`) without global resource caps, allowing multi-IP distributed attacks to bypass per-IP controls. The delay level logic relies on timing between consecutive requests but lacks sophisticated burst detection, enabling attackers to shape traffic patterns to avoid triggering backoff mechanisms while still overwhelming system resources through coordinated flooding from multiple source addresses.

Proof of Concept

```
public static async Task<T> SignalRQueue<T>(HubCallerContext context, int sizeCost, Func<Task<T>> func)
{
    var now = TimeUtil.GetMillisecondTime();
    var ipAddress = GetIP(context);
    ...
    if (Globals.MessageLocks.TryGetValue(ipAddress, out var Lock))
    {
        var prev = Interlocked.Exchange(ref Lock.LastRequestTime, now);
        if (Lock.ConnectionCount > 20)
            BanService.BanPeer(ipAddress, ipAddress + ": Connection count exceeded limit.  Peer failed to wait for

        if (Lock.BufferCost + sizeCost > 5000000)
        {
            throw new HubException("Too much buffer usage.  Message was dropped.");
        }
        if (now - prev < 1000)
            Interlocked.Increment(ref Lock.DelayLevel);
        else
        {
            Interlocked.CompareExchange(ref Lock.DelayLevel, 1, 0);
            Interlocked.Decrement(ref Lock.DelayLevel);
        }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Implement global resource caps for total buffer usage and concurrent requests across all connections to prevent distributed attacks from overwhelming system capacity.
• Add sliding-window rate limiting algorithms that can detect sophisticated burst patterns and coordinated multi-IP attacks more effectively than simple timing-based approaches.
• Establish per-role resource budgets and dynamic threshold adjustment based on system load metrics, with comprehensive monitoring and alerting for rate limiting events.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/66dfc000c7726407763abc399331bdf765e62642

7.42 TRANSACTION NONCE ORDERING NOT ENFORCED

// MEDIUM

Description

The transaction validation pipeline does not enforce that `tx.Nonce` equals the sender's expected account nonce, accepting transactions with any nonce value as long as signature and balance checks pass. During state application via `StateData.UpdateTreis`, the sender's nonce is unconditionally incremented regardless of the transaction's nonce value. This design permits out-of-order execution and duplicate-nonce acceptance across different blocks, weakening replay protection beyond identical-hash detection and enabling subtle transaction ordering attacks that can affect application-level logic expecting monotonic nonce sequences.

Proof of Concept

```
var newTxn = new Transaction()
{
    Timestamp = txRequest.Timestamp,
    FromAddress = txRequest.FromAddress,
    ToAddress = txRequest.ToAddress,
    Amount = txRequest.Amount,
    Fee = txRequest.Fee,
    Nonce = txRequest.Nonce,
    TransactionType = txRequest.TransactionType,
    Data = txRequest.Data,
    UnlockTime = txRequest.UnlockTime,
};

//Signature Check - Final Check to return true.
if (!string.IsNullOrEmpty(txRequest.Signature))
{
    var isTxValid = SignatureService.VerifySignature(txRequest.FromAddress, txRequest.Hash, txRequest.Signature);
    if (isTxValid)
        txResult = true;
    else
        return (txResult, "Signature Failed to verify.");
}

var from = GetSpecificAccountStateTrei(tx.FromAddress);
// ...
from.Nonce += 1;
from.StateRoot = block.StateRoot;
from.Balance -= (tx.Amount + tx.Fee);
accStTrei.UpdateSafe(from);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

• Implement strict nonce validation by enforcing `tx.Nonce == expectedNonce(sender)` during transaction verification and rejecting transactions with stale or future nonce values beyond acceptable windows.

• Maintain per-account pending nonce tracking in the mempool to prevent admission of transactions that cannot be immediately processed due to nonce gaps or conflicts.
• Establish nonce usage recording in state transitions or derive validation from expected nonce calculations to prevent duplicate-nonce acceptance across different blocks and maintain transaction ordering integrity.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/5c8fc5656c2d152b88d75e0529f05d7126c10776

7.43 RESERVE CALLBACK/RECOVER LACK IDEMPOTENCE; LOCKED BALANCE UNDERFLOW RISK

// MEDIUM

Description

Reserve flows (confirmation, callback, recovery) update balances and locked balances without idempotence guards and without lower-bound checks. Replaying these handlers (or concurrent re-entry) can double-apply state transitions, drive `LockedBalance` negative, or mis-account funds. Upstream validation mitigates duplicate CallBack by checking `ReserveTransactionStatus == Pending`, but there is a crash window (after state updates, before status persistence) that permits re-application. For Recover, upstream validation does not gate on status, so repeats are easier. State application still needs idempotent, bounded updates.

Upstream validation context (mitigations and gaps)
- CallBack(): validation rejects repeats when status != Pending.

```
if (rTx.ReserveTransactionStatus != ReserveTransactionStatus.Pending)

    return (txResult, $"This TX already has a status of: {rTx.ReserveTransactionStatus}");
```

- However, state code applies decrements before setting status to CalledBack; a crash between these steps permits a second valid CallBack to re-apply decrements.
- Recover(): validation verifies recovery signature/time but does not check/respect existing `ReserveTransactionStatus`, allowing repeated Recover attempts to re-apply decrements.

Proof of Concept

```
private static void CallBackReserveAccountTx(string? callBackHash)
{
    ...
    if(rTX.TransactionType == TransactionType.TX)
    {
        var stateTreiFrom = GetSpecificAccountStateTrei(rTX.FromAddress);
        var stateTreiTo = GetSpecificAccountStateTrei(rTX.ToAddress);
        if (stateTreiFrom != null)
        {
            stateTreiFrom.LockedBalance -= rTX.Amount; // no floor, no idempotence guard
            stateTreiFrom.Balance += rTX.Amount;
            stDb.UpdateSafe(stateTreiFrom);
            ...
            rLocalAccount.LockedBalance -= rTX.Amount; // no floor
            rDb.UpdateSafe(rLocalAccount);
        }
        if (stateTreiTo != null)
        {
            stateTreiTo.LockedBalance -= rTX.Amount; // no floor
            stDb.UpdateSafe(stateTreiTo);
            ...
            localAccount.LockedBalance -= rTX.Amount; // no floor
            accountDB.UpdateSafe(localAccount);
            ...
            rLocalAccount.LockedBalance -= rTX.Amount; // no floor
            rDb.UpdateSafe(rLocalAccount);
        }
```

```
    }
    ...
    // sets status to CalledBack but does not check current status before applying
    localTx.TransactionStatus = TransactionStatus.CalledBack;
    rTX.ReserveTransactionStatus = ReserveTransactionStatus.CalledBack;
    ...
}
```

Recover iterates all reserve TX for the address and applies balance moves without prior status gating or floors:

```
private static void RecoverReserveAccountTx(string? _recoveryAddress, string _fromAddress, string stateRoot)
{
    var rTXList = ReserveTransactions.GetTransactionList(_fromAddress);
    foreach(var rTX in rTXList)
    {
        if (rTX.TransactionType == TransactionType.TX)
        {
            stateTreiFrom.LockedBalance -= rTX.Amount; // no floor, no status/idempotence guard
            ...
            stateTreiTo.LockedBalance -= rTX.Amount; // no floor
            ...
        }
        if (rTX.TransactionType == TransactionType.NFT_TX)
        {
            scStateTrei.NextOwner = null; scStateTrei.IsLocked = false; // no status gating
        }
        ...
        localTx.TransactionStatus = TransactionStatus.Recovered;
        rTX.ReserveTransactionStatus = ReserveTransactionStatus.Recovered;
        rtxDb.UpdateSafe(rTX);
    }
    ...
    // bulk reset of balances without floors
    rsrvAccount.Balance = 0.0M;
    rsrvAccount.LockedBalance = 0.0M;
    stDb.UpdateSafe(rsrvAccount);
}
```

Reserve confirmation path updates balances without clear idempotence checks either:

```
public static void UpdateTreiFromReserve(List<ReserveTransactions> txList)
{
    foreach(var rtx in txList)
    {
        if (rtx.TransactionType == TransactionType.TX)
        {
            if (rtx.FromAddress.StartsWith("xRBX"))
            {
                to.Balance += rtx.Amount;
                to.LockedBalance -= rtx.Amount; // no floor
                accStTrei.UpdateSafe(to);
            }
        }
        ...
        rtx.ReserveTransactionStatus = ReserveTransactionStatus.Confirmed; // no skip if already confirmed
        rtxDb.UpdateSafe(rtx);
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Enforce idempotence per transaction in state handlers:
- Before any mutation, load the current `ReserveTransactionStatus` / `TransactionStatus` ; skip if already in a terminal state (Confirmed, CalledBack, Recovered).
- For `RecoverReserveAccountTx` , filter to entries with `ReserveTransactionStatus == Pending` (or a dedicated recoverable state), not all history.
- Use a “mark-then-apply-then-finalize” pattern or atomic flag to remove the crash window.
- Add lower-bound checks: prevent negative `LockedBalance` / `Balance` (clamp to zero and log, or reject and roll back).
- Wrap multi-record updates and status transitions in a DB transaction; commit only if all mutations succeed.
- Add startup reconciliation that replays pending reserve operations until statuses and balances are consistent.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/842485e53b1dfd4352c0705fcde2438ba5125940

7.44 MISSING FEE FLOOR AND GLOBAL MEMPOOL LIMITS ENABLE ECONOMIC/SPACE DOS

// MEDIUM

Description

Nodes accept transactions into the mempool based on rating and basic validity, but there is no enforced fee floor, and no hard global mempool size/entry cap. While `MempoolSizeUtility` limits block assembly to ~1 MB, the in-memory/on-disk mempool can grow unbounded and be populated with very low-fee transactions, enabling sustained resource pressure and propagation overhead.

Proof of Concept

Evidence

- No fee floor checks found in transaction validation or insertion paths.
- Mempool insertion occurs when rating/double-spend checks pass; no global size cap.

```
if (dblspndChk == false && isCraftedIntoBlock == false && rating != TransactionRating.F)

{

    mempool.InsertSafe(transaction);

}
```

- Block assembly only trims to 1 MB; it does not prune mempool growth.

```
var mempoolSize = Convert.ToInt64(currentSize + size).ToSize(...);

if (mempoolSize > 1.00M) return (false, 0);
```

- Rating service has per-address heuristics (e.g., count thresholds) but no mempool-wide cap or fee-based prioritization.

```
if (mempool.Count() > 25) { ... }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Enforce a dynamic or static fee floor for mempool admission (reject below threshold).
- Add a global mempool entry/size cap (maximum N entries or M MB). On overflow, evict by priority (fee rate, age, rating), not FIFO.

- Consider per-origin (address/IP) quotas coupled with rating to prevent one sender from monopolizing capacity.
- Keep `MempoolSizeUtility` for block assembly but also prune mempool periodically to target bounds.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/4f5784ab47c48942d0e5ca8602f8a3e5cdf9bb09

7.45 UNBOUNDED DESERIALIZATION AND MISSING CANCELLATION/BACKPRESSURE IN NODEDATAPROCESSOR

// MEDIUM

Description

`NodeDataProcessor.ProcessData` deserializes blocks and transactions from arbitrary peers without local size checks, cancellation, or explicit backpressure. While upstream SignalRQueue DoS guards exist at server entrypoints, the client-side processor should still defensively validate payload sizes and support cancellation to mitigate memory/CPU spikes from large or bursty inputs.

Proof of Concept

```
public static async Task ProcessData(string message, string data, string ipAddress)
{
    ...
    if (message == "blk")
    {
        var nextBlock = JsonConvert.DeserializeObject<Block>(data); // no size cap or cancellation
        ...
    }
    if(message == "7777")
    {
        var transaction = JsonConvert.DeserializeObject<Transaction>(data); // no size cap or cancellation
        ...
        mempool.InsertSafe(transaction);
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

- Enforce per-message size caps before deserialization (1–2 MB blocks, 64–256 KB tx), reject above limits.
- Use `JsonTextReader` with `MaxDepth` and streaming limits, or a pre-parse length check.
- Thread a `CancellationToken` through ProcessData and honor it in any async waits.
- Add lightweight per-IP rate accounting at this layer (cheap counters) to early-drop bursts.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/3684968b6f022080cdc6ca83ec7907f3b9a8d0ed

7.46 SIGNATURE GENERATION LACKS VALIDATION FOR ZERO COMPONENTS

// LOW

Description

The `sign` method in the `Ecdsa` class does not validate that the generated signature components `r` and `s` are non-zero before returning the signature. According to ECDSA standards, valid signatures must satisfy `0 < r < N` and `0 < s < N`. While the `verify` method correctly rejects signatures with zero components by checking `sigR < 1` and `sigS < 1`, the signing process can produce such invalid signatures under rare circumstances.

The value `r` becomes zero when `randSignPoint.x ≡ 0 (mod N)`, which occurs when the x-coordinate of the random point equals the curve order `N` modulo the prime `P`. For secp256k1, this happens when `randSignPoint.x = N`, corresponding to specific points on the curve. The value `s` becomes zero when `(numberMessage + r * privateKey.secret) ≡ 0 (mod N)`, which can occur even with non-zero `r` values depending on the message hash and private key combination.

Although these occurrences are extremely rare, an honest signer following the current implementation could generate signatures that are immediately rejected by any compliant verifier, potentially causing transaction failures or authentication issues in production systems.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Implement a retry loop in the `sign` method that continues generating new random nonces until both `r` and `s` are non-zero. After computing the signature components, check if either `r.IsZero` or `s.IsZero` and regenerate the signature with a new random nonce if so. This ensures all generated signatures comply with ECDSA requirements and will pass verification, eliminating the risk of producing invalid signatures during normal operation.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/10

7.47 SIGNATURE VERIFICATION ACCEPTS OFF-CURVE PUBLIC KEYS

// LOW

Description

The `verify` method in the `Ecdsa` class does not validate that the provided public key point lies on the elliptic curve before performing signature verification. This omission allows attackers to supply arbitrary off-curve points as public keys, which can potentially compromise the security assumptions of ECDSA. When a public key point does not satisfy the curve equation `y² = x³ + ax + b (mod p)`, the mathematical operations during verification may produce predictable or exploitable results.
An attacker could craft specific off-curve public key coordinates that, when combined with carefully constructed signature values, might pass the verification check even without knowledge of the corresponding private key. This violates the fundamental security property that only the holder of the private key should be able to generate valid signatures. The verification process assumes the public key is mathematically valid and on the curve, but this assumption is not enforced by the current implementation.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Add a curve membership validation check at the beginning of the `verify` method before performing any cryptographic operations. The verification should call `curve.contains(publicKey.point)` and return `false` immediately if the public key point does not lie on the specified elliptic curve. This validation ensures that only mathematically valid public keys are accepted for signature verification, maintaining the security guarantees of the ECDSA algorithm.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/11

7.48 SYNCHRONOUS DISPOSAL BLOCKS ON ASYNCHRONOUS OPERATIONS CAUSING POTENTIAL HANGS

// LOW

Description

Multiple P2P client classes implement a synchronous `Dispose(bool disposing)` method that blocks on asynchronous disposal operations using the sync-over-async anti-pattern. The affected classes `ConsensusClient`, `P2PValidatorClient`, and `P2PClient` all iterate through node connections and call `node.Connection.DisposeAsync().ConfigureAwait(false).GetAwaiter().GetResult()`, which synchronously blocks the calling thread while waiting for asynchronous disposal operations to complete.

This pattern creates significant operational risks during node shutdown scenarios. When the synchronous `Dispose()` method is called, it will block the current thread until all peer connections have completed their asynchronous disposal, which can take unpredictable amounts of time depending on network conditions, peer responsiveness, and connection states. If any peer connection disposal hangs or takes excessive time, the entire node shutdown process becomes blocked, potentially preventing graceful restarts, causing resource leakage, or requiring forceful termination.

The issue is particularly problematic because each class properly implements `DisposeAsyncCore()` methods that correctly handle asynchronous disposal, but the synchronous disposal path creates a deadlock-prone fallback that defeats the purpose of the async implementation. During high-stress network conditions or when peers are unresponsive, this can significantly impact node availability and operational reliability.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- Replace the blocking synchronous disposal with lightweight cleanup that does not wait for network operations.
- Modify the `Dispose(bool disposing)` method to perform only immediate, non-blocking cleanup operations such as setting cancellation tokens or marking connections for disposal. Ensure that the primary disposal logic remains in `DisposeAsyncCore()` and that calling code consistently uses `DisposeAsync()` for proper cleanup.
- Consider adding timeout mechanisms and error handling to the async disposal path to prevent indefinite blocking during network issues, and implement structured logging to monitor disposal performance and identify problematic peer connections.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/0e3ffdbcc85729c2bd96c330308592143cfc5177

7.49 MISSING DOS GUARD AND RATE LIMITS ON BLOCK RECEPTION

// LOW

Description

The contract `P2PValidatorServer` processes incoming blocks in the `ReceiveBlockVal` method without any denial-of-service (DoS) guard, such as a `SignalRQueue`, per-call byte budget, or rate limiting. This means that large or frequent block submissions can trigger significant serialization, validation, and broadcast work, enabling attackers to exhaust CPU, memory, and network resources. The absence of early rejection for oversized blocks and lack of per-IP or per-height rate controls increases the risk of resource exhaustion, slows down consensus, and degrades peer responsiveness. Amplification effects are possible when malicious blocks are broadcast to other peers.

Proof of Concept

- `ReceiveBlockVal` accepts and processes blocks without wrapping the handler in a `SignalRQueue` or enforcing any per-call or per-IP limits.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- Wrap `ReceiveBlockVal` with `SignalRQueue(Context, nextBlock.Size + overhead, ...)` and enforce per-IP rate limits and height windows.
- Reject oversized blocks early in the handler and apply backpressure or cooldowns for abusive sources.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/1dfd18e62e433f0652d83185a6e3b308e128cb9e

7.50 MISSING PRE-VALIDATION FILTERS ON BLOCK RECEPTION

// LOW

Description

The contract `P2PValidatorServer` processes incoming blocks in the `ReceiveBlockVal` method without performing fast pre-validation checks on block headers, size, version, timestamp drift, or parent hash existence before accepting and processing the block. Only basic checks such as `ChainRefId` and simple height comparisons are performed before insertion or broadcast decisions. This omission allows obviously invalid, far-ahead, or malformed blocks to consume CPU and memory resources, increasing the risk of slowdowns for legitimate processing, especially under load.

Proof of Concept

- `ReceiveBlockVal` does not check block size, version, timestamp, or parent hash before processing and broadcasting.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- Add checks for maximum block size, supported version, timestamp drift, and parent/previous hash existence window before queuing or processing the block.
- Maintain a small cache of recently seen or invalid block hashes to quickly drop repeated submissions.
- Defer serialization and broadcast until all pre-validation checks have passed.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/1a29d87929c622e77c146575040596da81481d28

7.51 PREMATURE EXIT FROM MAJORITY CALCULATION

// LOW

Description

In the consensus contract, the logic for calculating the majority threshold and wait-sets relies on volatile peer metadata and a fixed `HeartBeatTimeout`. Edge cases, such as peers toggling their `IsFinalized` status or clock skew in `LastMethodCodeTime`, can trigger early termination paths that return `null` even when a majority may form shortly after. This can result in fragile progress under network churn, degrading consensus participation and increasing variance in finalization time.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
var WaitForAddresses = AddressesToWaitFor(Height, methodCode, HeartBeatTimeout);
while (Height == Globals.LastBlock.Height + 1)
{
    var RemainingAddressCount = !ConsensusSource.IsCancellationRequested ? WaitForAddresses.Except(Messages.Selec
    if ((methodCode != 0 && Messages.Count + RemainingAddressCount < Majority) ||
        (RemainingAddressCount == 0 && Messages.Count >= Majority))
        break;
    await Task.Delay(20);
    WaitForAddresses = AddressesToWaitFor(Height, methodCode, HeartBeatTimeout);
}
...
if (Messages.Count >= Majority)
{
    ConsensusSource.Cancel();
    break;
}
...
return null; // not enough messages
```

This logic can exit prematurely if peer metadata changes or timeouts occur, even though a majority may be achievable with slightly more waiting.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- Recompute majority based on live membership to ensure the threshold reflects current network conditions.
- Include damping or hysteresis in the calculation to avoid reacting to transient peer state changes.
- Add a minimum observation period before terminating to reduce the risk of premature exits.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/aab9e83e1aab92f4b927b88a66ac78c5daeac860

7.52 SYNCHRONOUS BLOCKING ON ASYNC DISPOSAL

// LOW

Description

In the contract handling SignalR connections, the `Dispose(bool)` method synchronously blocks on `DisposeAsync()` for each node connection using `GetAwaiter().GetResult()`. This pattern can cause deadlocks or significantly increase shutdown latency, especially under load or in UI contexts. The contract provides an async disposal path (`DisposeAsyncCore()`), which should be used to avoid resource contention and improve teardown reliability.

Proof of Concept

The following code from the contract demonstrates the issue:

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        foreach(var node in Globals.Nodes.Values)
            if(node.Connection != null)
                node.Connection.DisposeAsync().ConfigureAwait(false).GetAwaiter().GetResult();
    }
}
```

This synchronous blocking can lead to deadlocks or long pauses during resource cleanup.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- Remove synchronous blocking and route disposal through the async `DisposeAsync()` path.
- Collect disposal tasks and use `await Task.WhenAll` to ensure proper asynchronous teardown without blocking the main thread.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/0e3ffdbcc85729c2bd96c330308592143cfc5177

7.53 UNBOUNDED GROWTH OF MESSAGE/HASH CACHES

// LOW

Description

In the consensus contract, the `Messages` and `Hashes` caches are pruned opportunistically but lack global hard caps or TTL-based eviction on key cardinality. Under pathological conditions, such as many heights or methods from faulty peers, these caches can grow indefinitely, increasing memory pressure and risking performance degradation. The current pruning logic only removes some keys and does not enforce overall limits, leaving the contract vulnerable to gradual memory growth.

Proof of Concept

The following code from the consensus contract demonstrates the issue:

```
public static ConcurrentDictionary<(long Height, int MethodCode), ConcurrentDictionary<string, (string Message, st
public static ConcurrentDictionary<(long Height, int MethodCode), ConcurrentDictionary<string, (string Hash, strin
```

Pruning is performed as follows, but without global caps:

```
public static void RemoveStaleCache(NodeInfo node)
{
    var NodeHeight = node.NodeHeight + 1;
    var MessageKeys = ConsensusServer.Messages.Where(x => x.Value.ContainsKey(node.Address)).Select(x => x.Key).To
    var HashKeys = ConsensusServer.Hashes.Where(x => x.Value.ContainsKey(node.Address)).Select(x => x.Key).ToHashS
    ...
    foreach (var key in MessageKeys.Where(x => x.MethodCode != node.MethodCode && state.MethodCode != x.MethodCode
        if (ConsensusServer.Messages.TryGetValue(key, out var message))
            message.TryRemove(node.Address, out _);
    foreach (var key in HashKeys.Where(x => (node.IsFinalized && x.MethodCode != node.MethodCode) ||
        (!node.IsFinalized && x.MethodCode + 1 == node.MethodCode)))
        if (ConsensusServer.Hashes.TryGetValue(key, out var hash))
            hash.TryRemove(node.Address, out _);
}
```

Without hard caps or TTL, adversarial input can cause unchecked cache growth.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

- Add TTL and hard caps, such as maximum tracked heights/methods and maximum entries per key, dropping the oldest entries when limits are exceeded.
- Implement periodic background compaction and instrument cache size metrics and alarms to monitor and control memory usage.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f35e3547b90f3a372908240943012471661c241c

7.54 PRE-AUTHENTICATION STATE AND BALANCE VALIDATION

// LOW

Description

The connection handler in the validator service performs expensive trie lookups and balance verification before authenticating the connecting peer's signature. The `GetSpecificAccountStateTrei` method is called to retrieve account state, followed by balance validation against `ValidatorRequiredAmount()`, all occurring prior to signature verification via `SignatureService.VerifySignature`. This design flaw allows unauthenticated attackers to force the server to perform database operations and potentially leak information about address existence and balance status through timing side-channels and different error response patterns.

Proof of Concept

```
var stateAddress = StateData.GetSpecificAccountStateTrei(address);
if(stateAddress == null)
{
    _ = EndOnConnect(peerIP, "X", startTime, conQueue,
        "Connection Attempted, But failed to find the address in trie. You are being disconnected.",
        "Connection Attempted, but missing field Address: " + address + " IP: " + peerIP);
    return;
}

if(stateAddress.Balance < ValidatorService.ValidatorRequiredAmount())
{
    _ = EndOnConnect(peerIP, "W", startTime, conQueue,
        $"Connected, but you do not have the minimum balance of {ValidatorService.ValidatorRequiredAmount()} VFX.
        $"Connected, but you do not have the minimum balance of {ValidatorService.ValidatorRequiredAmount()} VFX:
    return;
}

var verifySig = SignatureService.VerifySignature(address, SignedMessage, signature);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Restructure the authentication flow to verify the signature first using `SignatureService.VerifySignature` before performing any state or balance validation operations. • Implement rate limiting and backpressure mechanisms on pre-authentication code paths to prevent resource exhaustion attacks from unauthenticated peers. • Standardize error messages and response timings across all failure scenarios to eliminate information disclosure through behavioral side-channels that could reveal address existence or balance status.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/7ecfc83d937b24836919cd827209628dbe01c221

7.55 SIGNATURE REUSE MAP GROWS WITHOUT CLEANUP

// LOW

Description

The connection handler inserts provided signatures into the `Globals.Signatures` map before performing signature verification, creating a memory exhaustion vector. The `TryAdd` operation occurs prior to authentication via `SignatureService.VerifySignature`, and failed signatures are never removed from the map. Attackers can exploit this by sending unique invalid signatures in connection attempts, causing unbounded growth of the signature tracking map and potential performance degradation over time.

Proof of Concept

```
if (!Globals.Signatures.TryAdd(signature, Now))
{
    await EndOnConnect(peerIP, "40", startTime, conQueue, "Reused signature.", "Reused signature.");
    return;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Modify the signature verification flow to insert entries into `Globals.Signatures` only after successful signature verification using `SignatureService.VerifySignature`. • Implement time-to-live (TTL) mechanisms and hard capacity limits on the signature map to prevent unbounded memory growth. • Add cleanup procedures to remove signature entries when authentication fails, ensuring failed connection attempts do not permanently consume memory resources.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/1de47fb07780a40b035bb294094a3b314b873bd0

7.56 UNSAFE PARSING OPERATIONS IN TASK ANSWER PROCESSING

// LOW

Description

The `ReceiveTaskAnswerV3` handler performs unsafe parsing operations on untrusted input without proper validation guards. The method splits the `request` parameter and directly uses `int.Parse` and `long.Parse` on the resulting array elements without verifying the parsed values are within acceptable ranges. While a 30-character length limit exists as a basic protection mechanism, malformed numeric inputs can still cause format exceptions that rely on broad catch-all error handling. This approach degrades system throughput under malicious input and obscures root causes of parsing failures, potentially enabling log pollution attacks.

Proof of Concept

```
var ipAddress = GetIP(Context);
if (request?.Length > 30)
{
    BanService.BanPeer(ipAddress, "request too big", "ReceiveTaskAnswerV3");
    return new TaskAnswerResult { AnswerCode = 5 };
}
...
var taskResult = request?.Split(':');
if (taskResult == null || taskResult.Length != 2)
{
    taskAnsRes.AnswerCode = 5;
    return taskAnsRes;
}

var (Answer, Height) = (int.Parse(taskResult[0]), long.Parse(taskResult[1]));
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Replace unsafe parsing methods with `int.TryParse` and `long.TryParse` operations that include explicit bounds checking to validate numeric ranges and reject invalid values with clear error codes. • Implement more robust input validation using structured data formats such as JSON with schema validation to ensure data integrity and type safety. • Add per-IP rate limiting mechanisms on parsing failures to prevent abuse and reduce the impact of malformed input attacks on system performance.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/0b0134bf16022a1c9384cc29d2f793c911aa05b7

7.57 UNBOUNDED JSON INPUTS AND UNSAFE ASSET NAME HANDLING

// LOW

Description

Multiple endpoints perform unsafe deserialization of untrusted JSON data without implementing size limits or validation checks. The `JsonConvert.DeserializeObject<string[]>(data)` operation processes client-provided arrays without constraining payload size, creating potential for memory exhaustion attacks. Additionally, the `assetName` field extracted from deserialized data is directly used in file operations such as `BeaconService.DeleteFile(beaconData.AssetName)` without proper sanitization or path normalization. This combination enables both denial-of-service attacks through oversized payloads and potential path traversal vulnerabilities where attackers can target files outside the intended directory structure.

Proof of Concept

```
var payload = JsonConvert.DeserializeObject<string[]>(data);
var scUID = payload[0];
var assetName = payload[1];

if(beaconData.DeleteAfterDownload)
    BeaconService.DeleteFile(beaconData.AssetName);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Implement explicit size and count limits before JSON deserialization operations to prevent memory exhaustion attacks from oversized payloads.
• Validate all `assetName` values against a strict whitelist or apply robust path normalization to prevent directory traversal and ensure only authorized files can be accessed.
• Configure file operations to execute within a sandboxed directory structure with restricted permissions to limit the impact of potential path manipulation attacks.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/e29ea7947e22f0c2d201d5786dc0e0b1f5734c5c

7.58 BLOCKCASTER HANDSHAKE: PRE-AUTHENTICATION STATE/BALANCE CHECKS

// LOW

Description

The `OnConnectedAsync` method performs expensive database operations before authenticating the connecting peer, creating opportunities for resource exhaustion and information disclosure. The function calls `StateData.GetSpecificAccountStateTrei(address)` to perform trie lookups and validates minimum balance requirements via `ValidatorService.ValidatorRequiredAmount()` comparison before executing signature verification. This sequence allows unauthenticated attackers to force the server to perform costly database operations and potentially infer sensitive information about address existence and balance status through timing analysis and distinct error message patterns returned for different validation failures.

Proof of Concept

```
var stateAddress = StateData.GetSpecificAccountStateTrei(address);
if (stateAddress == null)
{
    _ = EndOnConnect(peerIP,
        "Connection Attempted, But failed to find the address in trie. You are being disconnected.",
        "Connection Attempted, but missing field Address: " + address + " IP: " + peerIP);
    return;
}

if (stateAddress.Balance < ValidatorService.ValidatorRequiredAmount())
{
    _ = EndOnConnect(peerIP,
        $"Connected, but you do not have the minimum balance of {ValidatorService.ValidatorRequiredAmount()} VFX.
        $"Connected, but you do not have the minimum balance of {ValidatorService.ValidatorRequiredAmount()} VFX:
    return;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Restructure the authentication flow to perform signature verification first before executing any database operations, ensuring only authenticated peers can trigger expensive trie lookups and balance checks.
• Implement comprehensive rate limiting on pre-authentication code paths to prevent unauthenticated peers from overwhelming the system with resource-intensive requests.
• Standardize error responses and timing patterns across all authentication failure scenarios to eliminate information leakage that could reveal address existence or balance information to unauthorized parties.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/7ecfc83d937b24836919cd827209628dbe01c221

7.59 UNSAFE PARSING AND SUBSTRING OOB

// LOW

Description

The client performs unsafe parsing operations on untrusted network responses without implementing proper bounds checking or validation. The `RequestMethodCode` response handling uses `Split(':')` followed by direct `long.Parse` and `int.Parse` operations without verifying array bounds or validating numeric formats. Additionally, wallet version processing applies `Substring(0,3)` operations without length validation, creating out-of-bounds exceptions when responses contain malformed data or insufficient characters. These parsing vulnerabilities can cause connection failures and degrade overall network liveness when malicious or corrupted responses are received.

Proof of Concept

```
var Response = await node.Connection.InvokeCoreAsync<string>("RequestMethodCode",
    new object[] { Globals.LastBlock.Height, state.MethodCode, state.Status == ConsensusStatus.Finalized },
    new CancellationTokenSource(10000).Token);
...
var remoteMethodCode = Response.Split(':');
...
node.NodeHeight = long.Parse(remoteMethodCode[0]);
node.MethodCode = int.Parse(remoteMethodCode[1]);
node.IsFinalized = remoteMethodCode[2] == "1";

if (walletVersion != null)
{
    peer.WalletVersion = walletVersion.Substring(0,3);
    node.WalletVersion = walletVersion.Substring(0,3);
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Implement comprehensive length validation before substring operations and use safe parsing methods like `TryParse` instead of direct `Parse` calls to handle malformed input gracefully.
• Add bounds checking for array access after split operations to ensure sufficient elements exist before indexing into response arrays.
• Validate version string formats and lengths before processing to prevent out-of-bounds exceptions and ensure consistent handling of version information across the network.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/533df68531844b8fa17f0544fa44283b0901e872

7.60 PAYLOAD FIELDS NOT CRYPTOGRAPHICALLY BOUND TO AUTHENTICATED IDENTITY

// LOW

Description

The `SendMotherData` method updates node state information based on payload data without cryptographically binding the submitted data to the authenticated session identity. The method deserializes `Mother.DataPayload` objects and directly updates `Globals.MothersKids` entries using `payload.Address` as the key, but does not verify that the submitting session has legitimate authority to update information for that specific address. This allows authenticated users to submit misleading data for arbitrary addresses if passwords are compromised or reused across multiple nodes, potentially corrupting monitoring data and operational visibility.

Proof of Concept

```
var payload = JsonConvert.DeserializeObject<Mother.DataPayload>(data);
if (payload != null)
{
    Globals.MothersKids.TryGetValue(payload.Address, out var kid);
    if (kid != null)
    {
        kid.Address = payload.Address;
        kid.IPAddress = peerIP;
        kid.Balance = payload.Balance;
        ...
        Globals.MothersKids[payload.Address] = kid;
    }
    else
    {
        Mother.Kids nKid = new Mother.Kids {
            Address = payload.Address,
            IPAddress = peerIP,
            Balance = payload.Balance,
            ...
        };
        Globals.MothersKids[payload.Address] = nKid;
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Implement cryptographic binding by requiring digital signatures over all payload fields using the account's private key before accepting state updates.
• Add signature verification logic to ensure that only the legitimate owner of an address can submit data updates for that specific account.
• Establish address-to-session mapping validation to prevent cross-account data injection and maintain data integrity across the monitoring system.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/bec76a118bf8d452d6f2b8c199928e20ad0c360b

7.61 TOCTTOU IN PER-IP QUEUE ACCOUNTING (CONNECTIONCOUNT/BUFFERCOST)

// LOW

Description

The `SignalRQueue` implementation contains a time-of-check-time-of-use vulnerability in its accounting logic for connection counts and buffer costs. The counters are incremented before acquiring the semaphore and decremented immediately after semaphore entry, while the actual work function executes outside the accounting scope. This creates a window where the accounting metrics misrepresent the actual concurrent load, allowing brief overruns of the intended limits. During high contention periods, multiple requests can pass the initial threshold checks before the counters are properly updated, leading to temporary violations of the configured limits and reduced effectiveness of the backpressure mechanisms.

Proof of Concept

```
private static async Task<T> SignalRQueue<T>(MessageLock Lock, int sizeCost, Func<Task<T>> func)
{
    Interlocked.Increment(ref Lock.ConnectionCount);
    Interlocked.Add(ref Lock.BufferCost, sizeCost);
    T Result = default;
    try
    {
        await Lock.Semaphore.WaitAsync();
        Interlocked.Decrement(ref Lock.ConnectionCount);
        Interlocked.Add(ref Lock.BufferCost, -sizeCost);

        var task = func();
        if (Lock.DelayLevel == 0)
            return await task;
        ...
        Result = await task;
    }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Restructure the accounting logic to hold counter increments until after semaphore acquisition to ensure accurate representation of concurrent load during execution.
• Implement proper cleanup by decrementing counters only after the work function completes, maintaining accurate metrics throughout the entire request lifecycle.
• Add comprehensive try-finally safeguards around all counter operations to ensure proper cleanup even when exceptions occur during request processing.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/654f7fa2877a7a7b368217d397a13707a2d215cd

7.62 TRANSACTION STALENESS CHECK DEPENDS ON DOWNLOAD STATE

// LOW

Description

The transaction timestamp staleness validation, which rejects transactions older than approximately 60 minutes, only executes when both `Globals.BlocksDownloadSlim.CurrentCount` and `Globals.BlocksDownloadV2Slim.CurrentCount` are non-zero. This coupling to download semaphore states creates inconsistent validation behavior where stale transactions can bypass temporal checks depending on the node's current download activity. The conditional gating makes transaction acceptance dependent on unrelated blockchain synchronization operations, potentially allowing very old signed transactions to be accepted during certain runtime conditions.

Proof of Concept

```
//Timestamp Check
if (Globals.BlocksDownloadSlim.CurrentCount != 0 && Globals.BlocksDownloadV2Slim.CurrentCount != 0)
{
    var currentTime = TimeUtil.GetTime();
    var timeDiff = currentTime - txRequest.Timestamp;
    var minuteDiff = timeDiff / 60M;
    if (minuteDiff > 60.0M)
        return (txResult, "The timestamp of this transactions is too old to be sent now.");
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

• Remove semaphore gating logic and implement unconditional staleness validation in `VerifyTX` to ensure consistent temporal checks regardless of download state.
• Centralize timing thresholds in configuration parameters such as `MaxTxAgeSeconds=3600` and `MaxFutureSkewSeconds=120` and implement reusable validation via `TransactionData.IsTxTimestampStale` methods.
• Apply uniform staleness validation across all transaction admission paths including P2P, API, and wallet interfaces before mempool insertion, with appropriate logging and rejection handling for stale transactions.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/f8df1aa6fadfe20bf14f9ca75d5f9458543103d2

7.63 DOCUMENTATION CONTAINS SPELLING ERRORS AND MISLEADING DESCRIPTIONS

// INFORMATIONAL

Description

The `EcdsaMath` class contains multiple documentation issues that reduce code maintainability and clarity. The comments throughout the file contain recurring spelling errors such as "mutiply" and "multily" instead of "multiply". Additionally, several function descriptions incorrectly describe the mathematical operations being performed. The `multiply` and `jacobianMultiply` methods are documented as returning "Point that represents the sum of First and Second Point" when they actually perform scalar multiplication, not point addition. The documentation also uses Python-style parameter syntax with `//:param` notation, which is inconsistent with C# conventions and does not integrate with IntelliSense or XML documentation generation tools.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Correct all spelling errors throughout the documentation comments, replacing "mutiply" and "multily" with "multiply". Update function descriptions to accurately reflect their mathematical operations, particularly ensuring scalar multiplication functions describe point multiplication rather than addition. Replace the Python-style parameter documentation with proper C# XML documentation tags using `<summary>`, `<param>`, and `<returns>` elements to improve integration with development tools and provide better IntelliSense support for developers.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/pull/9/commits/d82893b9c474730bb4ea25bb4de75bf0133441bd

7.64 HARDCODED TIMEOUTS AND FIXED DELAYS WITHOUT OBSERVABILITY

// INFORMATIONAL

Description

The contract `P2PValidatorServer` uses hardcoded timeout values (such as `2000ms` and `6000ms`) and fixed delays in client notifications and block validation routines. For example, `new CancellationTokenSource(2000/6000)` is used in client notification calls, and `await Task.Delay(2000)` is used in `ReceiveBlockVal` before validation. These static values are not adaptive to real network conditions or system load, and there is no instrumentation or metrics to inform tuning. As a result, under varying network conditions or partial failures, these timeouts and delays may be too short or too long, leading to flakiness, avoidable retries, and reduced system observability.

Proof of Concept

- `new CancellationTokenSource(2000/6000)` in client notifications.
- `await Task.Delay(2000)` in `ReceiveBlockVal` before validation.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Centralize timeout configuration and expose these values via settings and metrics for easier tuning.
- Use exponential backoff with jitter for retries instead of fixed delays, especially on hot paths.
- Instrument notification and validation attempts/failures to drive adaptive tuning and improve observability.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/e2f16b6379964ea92b61f749ee6a3322205fb293

7.65 UNAUTHENTICATED VALIDATOR LIST UPDATES AND WEAK BINDING CHECKS

// INFORMATIONAL

Description

The contract `P2PValidatorServer` implements the `SendNetworkValidatorList` method, which accepts peer-supplied validator lists and updates the in-memory registry (`Globals.NetworkValidators`) without cryptographic attestation or schema and size constraints. The only binding check present is a weak test (`Signature.Contains(publicKey)`), which does not provide sound cryptographic assurance that the public key is actually bound to the address. There is also a TODO comment in the code (“TODO: ADD PROTECTION”) acknowledging the absence of proper safeguards. As a result, a single peer can inject arbitrary validator records, polluting the registry and potentially biasing peering topology. No limits are enforced on the size, depth, or schema of the incoming JSON, increasing the risk of resource abuse.

Proof of Concept

```
// Get Network Validator List - TODO: ADD PROTECTION
public async Task SendNetworkValidatorList(string data)
{
    ...
    var networkValList = JsonConvert.DeserializeObject<List<NetworkValidator>>(data);
    if(networkValList?.Count > 0)
    {
        foreach(var networkValidator in networkValList)
        {
            if(Globals.NetworkValidators.TryGetValue(networkValidator.Address, out var networkValidatorVal))
            {
                var verifySig = SignatureService.VerifySignature(
                    networkValidator.Address,
                    networkValidator.SignatureMessage,
                    networkValidator.Signature);

                //if(networkValidatorVal.PublicKey != networkValidator.PublicKey)

                if(verifySig && networkValidator.Signature.Contains(networkValidator.PublicKey))
                    Globals.NetworkValidators[networkValidator.Address] = networkValidator;
            }
            else
            {
                Globals.NetworkValidators.TryAdd(networkValidator.Address, networkValidator);
            }
        }
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Require per-entry cryptographic attestation: each advertised validator should sign a payload containing `(address, publicKey, ip, timestamp/nonce)` and verify the address–publicKey binding.
- Optionally, require corroboration by multiple trusted validators or an adjudicator-signed snapshot before accepting entries.
- Enforce input limits and schema validation: set maximum message size and depth, validate models strictly, and reject malformed entries.
- Rate-limit and cap updates per source; avoid persisting entries that fail attestation or freshness checks.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/ccf1393532eff8db5f128d7116ec53500f9989ec

7.66 UNBOUNDED VALIDATOR REGISTRY GROWTH WITHOUT PRUNING OR TTL

// INFORMATIONAL

Description

The contract maintains validator registry structures such as `Globals.NetworkValidators`, which are appended to from multiple sources—including handshake events and list updates—without explicit pruning, time-to-live (TTL), or deduplication beyond keying by address. As a result, stale or inactive validator entries can persist indefinitely unless overwritten externally. Over time, this leads to unbounded memory growth and inflates the payload sizes of endpoints that return validator lists, increasing CPU and bandwidth costs for all participants.

Proof of Concept

- `Globals.NetworkValidators.TryAdd(address, netVal)` on connect and during list ingestion, with no observed TTL or eviction policy.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Introduce TTL or eviction policies for inactive or stale validators, and periodically prune the registry.
- Track last-seen timestamps for each validator and remove entries that have not been updated within a defined horizon.
- Ensure that endpoints returning validator lists always paginate and cap response sizes, regardless of the total registry size.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/83f8ed368a06536c3bf78b817312b92bf63558e5

7.67 VRFNUMBER ENDIANNESS DEPENDENCY CAN CAUSE CROSS-PLATFORM CONSENSUS SPLITS

// INFORMATIONAL

Description

The V4 validation system uses `BitConverter.ToInt32` to derive `VRFNumber` values from SHA-256 hashes, creating a critical cross-platform compatibility issue. Since `BitConverter` behavior depends on system endianness, validators running on different architectures or .NET runtimes will compute different `VRFNumber` values from identical input data. This endianness dependency causes divergent sorting and selection of block producers, leading to persistent consensus splits when heterogeneous node environments are deployed. The issue affects all V4 validation paths and any producer selection flows that rely on `VRFNumber` ordering for winner determination.

Proof of Concept

```
public static async Task<(uint, string)> CreateProof(string address, string publicKey, long blockHeight, string pr
{
    // ...
    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] hashBytes = sha256.ComputeHash(combinedBytes);
        //Produces non-negative by shifting and masking
        int randomBytesAsInt = BitConverter.ToInt32(hashBytes, 0);
        uint nonNegativeRandomNumber = (uint)(randomBytesAsInt & 0x7FFFFFFF);
        vrfNum = nonNegativeRandomNumber;
        proof = ProofUtility.CalculateSHA256Hash(seed + vrfNum.ToString());
    }
    return (vrfNum, proof);
}

public static bool VerifyProof(string publicKey, long blockHeight, string prevBlockHash, string proofHash)
{
    // ...
    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] hashBytes = sha256.ComputeHash(combinedBytes);
        //Produces non-negative by shifting and masking
        int randomBytesAsInt = BitConverter.ToInt32(hashBytes, 0);
        uint nonNegativeRandomNumber = (uint)(randomBytesAsInt & 0x7FFFFFFF);
        vrfNum = nonNegativeRandomNumber;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

• Replace `BitConverter.ToInt32` with explicit big-endian byte interpretation to ensure consistent cross-platform behavior regardless of system architecture or runtime environment.

• Implement deterministic endian-agnostic derivation methods such as reading the first 8 bytes as big-endian `UInt64` or using full 256-bit comparison for ordering operations.
• Add comprehensive cross-platform test vectors and enforce canonical derivation functions across all implementations to prevent future consensus compatibility issues.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

NA

7.68 V4 PROOF VALIDATION LACKS COMMITTEE MEMBERSHIP AND WINNER ENFORCEMENT

// INFORMATIONAL

Description

The Version 4 block validation system accepts validator proofs based solely on cryptographic hash matching without enforcing committee membership or canonical winner selection. The `Version4Rules` method validates that a proof matches the hash derived from `(publicKey || blockHeight || prevBlockHash)` but does not verify that the signer is an authorized committee member for the current height or that the block producer was legitimately selected through the network's consensus process. This allows any party with a valid account and private key to self-construct consistent proofs and produce blocks out of turn, completely bypassing the intended leader election mechanism and enabling unauthorized block production.

Proof of Concept

```
public static async Task<(bool, string)> Version4Rules(Block block)
{
    try
    {
        var blockHeight = block.Height;
        var validatorAddress = block.Validator;
        var validatorProof = block.ValidatorAnswer;
        var validatorPubKey = block.ValidatorSignature.Split(".")[1];
        var pubKeyDecoded = HexByteUtility.ByteToHex(Base58Utility.Base58Decode(validatorPubKey));
        if (pubKeyDecoded.Length / 2 == 63) { pubKeyDecoded = "00" + pubKeyDecoded; }
        var pubKeyByte = HexByteUtility.HexToByte(pubKeyDecoded);
        var publicKey = PublicKey.fromString(pubKeyByte);
        var _PublicKey = "04" + ByteToHex(publicKey.toString());
        var isProofValid = await ProofUtility.VerifyProofAsync(_PublicKey, blockHeight, Globals.LastBlock.Hash, va
        var result = isProofValid ? "" : "Proof Invalid.";
        return (isProofValid, result);
    }

public static async Task<(uint, string)> CreateProof(string address, string publicKey, long blockHeight, string pr
{
    uint vrfNum = 0;
    var proof = "";
    string seed = publicKey + blockHeight.ToString() + prevBlockHash; // all public
    byte[] combinedBytes = Encoding.UTF8.GetBytes(seed);
    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] hashBytes = sha256.ComputeHash(combinedBytes);
        int randomBytesAsInt = BitConverter.ToInt32(hashBytes, 0);
        uint nonNegativeRandomNumber = (uint)(randomBytesAsInt & 0x7FFFFFFF);
        vrfNum = nonNegativeRandomNumber;
        proof = ProofUtility.CalculateSHA256Hash(seed + vrfNum.ToString());
    }
    return (vrfNum, proof);
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

• Implement committee membership validation by verifying that the block producer is included in the authorized signer set snapshot for the specific height before accepting any V4 blocks.
• Enforce canonical winner selection by validating that the producer corresponds to the network-agreed winner based on VRF values or other consensus-determined selection criteria.
• Replace the publicly computable proof construction with a proper VRF implementation that includes signer-set commitments and round metadata in the signed payload to prevent unauthorized block creation.

Remediation Comment

SOLVED: The **VerifiedX team** solved this issue.

Remediation Hash

https://github.com/VerifiedXBlockchain/VerifiedX-Core/commit/cb096dc7e6c2209f774e746b08a9ec2e6aded426
