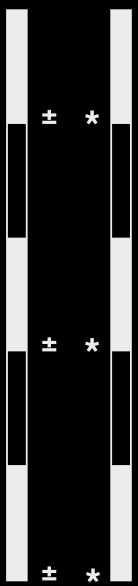
VERILOG X FUZZLAND
JOINT AUDIT

2024

NUBIT: SECURITY REVIEW REPORT



Sep 25, 2024

Nubit Audit Report

Executive Summary

From May 22, 2024, to Sep 25, 2024, Riema Labs engaged Fuzzland to conduct a thorough security audit of their Nubit Chain project. This audit focused on the various components of the Nubit Chain, including Nubit KZG, rsmt2d, Nubit Node, Nubit Validator, and Nubit Core. The primary objective was to identify and mitigate potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 80 person-days, involving 5 engineers who reviewed the code over a span of 16 days. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, Fuzzland team identified 20 issues across different severity levels and categories.

Project Summary

Nubit Chain is a scalable, cost-efficient data availability layer secured by Bitcoin, designed to enhance the Bitcoin ecosystem by addressing its data throughput limitations. It introduces innovations like a unique consensus mechanism (NubitBFT), Data Availability Sampling (DAS) for scalability, and a trustless Bitcoin-native bridge using the Lightning Network. Unlike other Bitcoin Layer 2 solutions focused on transaction scalability, Nubit emphasizes data availability to support applications like Ordinals, price oracles, and indexers while inheriting Bitcoin's security properties.

Scope

Project Name	Nubit KZG
Repository Link	https://github.com/RiemaLabs/nubit-kzg
Commit	aab930a56824dfe7b916209316ddda0b816df576
Language	Golang

Project Name	rsmt2d
Repository Link	https://github.com/RiemaLabs/rsmt2d
Commit	a53fa9453d059cdd5c43f123505649a270932817
Language	Golang

Project Name	Nubit Node
Repository Link	https://github.com/RiemaLabs/nubit-node
Commit	47f50350160bf5ddc202b37858505e990a256c53
Language	Golang

Project Name	Nubit Validator
Repository Link	https://github.com/RiemaLabs/nubit-validator
Commit	a44d4411a50caa8af3fbf230a2229205033257d6
Language	Golang

Project Name	Nubit Core
Repository Link	https://github.com/RiemaLabs/nubit-core
Commit	fc204be27227e99f49c32c5d421ce76dc91e229d
Language	Golang

Methodology

Our audit methodology comprised a comprehensive and systematic approach to uncover potential security vulnerabilities, risks, and coding issues. The key components of our methodology included:

- **Static Analysis**: We perform static analysis using our proprietary internal tools and CodeQL to identify potential vulnerabilities and coding issues.
- **Fuzz Testing**: We execute fuzz testing by utilizing our proprietary internal fuzzers to uncover potential bugs and logic flaws.
- **Formal Verification**: We develop individual tests for critical functions and leverage symbolic execution tools to prove the functions in question are not vulnerable.
- **Manual Code Review**: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Engagement Summary

The engagement involved a team of skilled consultants / engineers who were responsible for various phases of the audit process, including onboarding, initial audits, additional audits, and quality assurance. Below is a summary of the engagements with specific dates and details.

Dates	Consultants / Engineers Engaged	Details
5/22/2024	Qi Su, Eda Zhang	Onboarding
5/22/2024 - 6/14/2024	Qi Su, Taotao Zhou, Chao Wang, Tony Ke, Jacob Chia	Initial Audits
6/15/2024	Chaofan Shou, Eda Zhang	Quality Assurance
9/5/2024 - 9/24/2024	Qi Su, Taotao Zhou, Chao Wang, Tony Ke, Jacob Chia	Second Audits

Vulnerability Severity

We divide severity into four distinct levels: high, medium, low, and info. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as data breaches, system crashes, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.
- **Informational Severity Issues** represent informational findings that do not directly impact the system's security, functionality, or performance. These findings are typically observations or recommendations for potential improvements or optimizations. Addressing info severity issues can enhance the system's robustness and efficiency but is not necessary for the system's immediate operation or security. These issues can be considered for future development or enhancement plans.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	5	5
Medium Severity Issues	4	4
Low Severity Issues	4	4
Informational Severity Issues	7	7

Vulnerability Categories

We categorize issues into the following four distinct categories: Denial of Service (DOS), Logic Error (LOGIC), Consensus Issue (CONS), and Code Maintainability Issues (FORMAT). This classification allows for a more precise identification and remediation of specific types of vulnerabilities and flaws.

- **Denial of Service (DOS)** issues pertain to vulnerabilities that could enable an attacker to disrupt the normal operation of the software. These issues can lead to system outages, degraded performance, or complete service unavailability, thereby impacting the network's reliability and accessibility for legitimate users.
- **Logic Errors (LOGIC)** involve flaws in the code that cause the software to behave incorrectly or unpredictably. These errors can stem from incorrect assumptions, flawed algorithms, or improper handling of edge cases. Logic errors can lead to unintended actions, security breaches, or incorrect transaction processing, potentially undermining the integrity and reliability of the software.
- **Consensus Issues (CONS)** are vulnerabilities that could compromise the consensus mechanism. These issues can lead to disagreements among nodes about the state of the blockchain, resulting in forks, or other inconsistencies.
- **Code Maintainability Issues (FORMAT)** relate to aspects of the code that affect its readability, maintainability, and extensibility. These issues include poor documentation, inconsistent coding styles, and complex or redundant code structures. While they might not pose an immediate risk to the system's functionality or security, addressing maintainability issues is crucial for ensuring the long-term health and ease of future development and debugging of the blockchain code.

Below is a summary of the identified issues categorized by type, along with their current resolution status.

	Number	Resolved
Denial of Service (DOS)	4	4
Logic Errors (LOGIC)	4	4
Consensus Issues (CONS)	1	1
Code Maintainability Issues (FORMAT)	11	11

Disclaimer

The audit does not ensure that it has identified every security issue in the project, and it should not be seen as a confirmation that there are no more vulnerabilities. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value projects to commission several independent audits, a public bug bounty program, as well as continuous onchain security auditing and monitoring. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Nubit’s native token relies on the third-party library Cosmos-sdk. This audit treats third-party entities as a black box and assumes they function correctly. However, in reality, third-party components may be compromised, potentially rendering the Nubit chain unusable or vulnerable to damage.

Detailed Findings

[HIGH] Out-of-bound read in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
HIGH	DOS	go-kzg-4844/context.go	Fixed	ae7460a6

This issue arises within the `Context.DomainByIndex` function of the `nubit-kzg` project, where elements from the `domain.Roots` array are accessed via an index. The vulnerability occurs when an out-of-bounds index is used—either negative or exceeding the array's length—without proper checks in place beyond the current condition of the index being greater than `domain.Cardinality`. This flaw in index validation can cause an array out-of-bounds read, potentially leading to unpredictable software behavior or a system crash, which could be exploited to carry out a denial-of-service (DoS) attack. Although the system's architecture, based on Golang, primarily limits this to DoS without the possibility of remote code execution, the impact remains significant as it could disrupt service availability.

Relevant Code

```
// go-kzg-4844/context.go#12
func (c *Context) DomainByIndex(index int) (*fr.Element, error) {
  if index > int(c.domain.Cardinality) {
    return nil, ErrIndexOutOfRange
  }
  return &c.domain.Roots[index], nil
}
```

Proof of Concept (Exploit)

```
h := sha256.New()
kzg := NewKzgCommitter(h)
kzg.Open([][]byte{}, 4096)
// Or
kzg.Open([][]byte{}, -1)
```

Recommended Fix

Properly handle length validation at go-kzg-4844/context.go#12.

```
func (c *Context) DomainByIndex(index int) (*fr.Element, error) {
  if index >= int(c.domain.Cardinality) || index < 0 {
    return nil, ErrIndexOutOfRange
  }
  return &c.domain.Roots[index], nil
}
```

[HIGH] Out-of-bound read in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
HIGH	DOS	nubit-kzg/kzgproof.go	Fixed	160a56b2

This vulnerability pertains to six specific instances in the `kzgproof.go` file of the `nubit-kzg` project where slices are accessed using the syntax `v[:nIDLen]`. The function fails to ensure that `nIDLen` does not exceed the length of `v`, which can result in an out-of-bounds read. This can be triggered when the input data processed by these lines does not meet expected size conditions. An attacker can leverage this to cause a denial-of-service (DoS) condition by sending crafted inputs that lead to system instability or termination.

Relevant Code

```
// kzgproof.go#L262,L270
if !bytes.Equal(v[:nIDLen], nID) {
// kzgproof.go#L279,L289,L319,L318
if bytes.Equal(v[:nIDLen], nID) {
```

Proof of Concept (Exploit)

```
proof := NamespaceRangeProof{
  start:     0,
  end:       0,
  preIndex:  0,
  postIndex: 0,
  openStart: KzgOpen{
   value: make([]byte, 0),
  },
  openEnd:            KzgOpen{},
  openPreIndex:       KzgOpen{},
  openPostIndex:      KzgOpen{},
  InclusionOrAbsence: true,
}
proof.VerifyInclusion(nil, namespace.ID{0}, nil)
```

Recommended Fix

Properly handle length validation at nubit-kzg/kzgproof.go.

[HIGH] Out-of-bound write in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
HIGH	DOS	nubit-kzg/hasher.go	Fixed	160a56b2

The function `KzgHasher.Write` in the `nubit-kzg` project exhibits a critical vulnerability where it attempts to write to an array `e.data` without validating that there is sufficient allocated space. This oversight occurs during the copying of `leafHash` into `e.data`, utilizing the length of `leafHash` to dictate the end boundary of the copy operation without a preceding length check of `e.data`. This can result in memory being corrupted beyond the bounds of the allocated array, which can lead to application crashes or other unintended behaviors, ultimately facilitating a denial-of-service (DoS) scenario. Given the function's role within the system, this vulnerability underscores a critical oversight in data handling security.

Relevant Code

```
// nubit-kzg/hasher.go:L147
func (e *KzgHasher) Write(leafHash []byte) (n int, err error) {
  if len(leafHash)%gokzg4844.SerializedScalarSize != 0 {
    return 0, fmt.Errorf("leaf hash length is not a multiple of %d",
gokzg4844.SerializedScalarSize)
  }
  copy(e.data[e.length:len(leafHash)], leafHash)
  e.length = len(leafHash)
  return len(leafHash), nil
}
```

Proof of Concept (Exploit)

```
hasher := NewKzgHasher()
longLeaf := make([]byte, 131072+32)
hasher.Write(longLeaf)
```

Recommended Fix

Properly handle length validation at nubit-kzg/hasher.go:L147.

[HIGH] Out-of-bound write in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
HIGH	DOS	nubit-kzg/commit.go	Fixed	160a56b2

This bug is identified in the `KzgCommiter.Commit` method within the `nubit-kzg` project, where there is a failure to check the bounds before writing data into the `data` array. Specifically, the method does not verify whether the `leafHashes` array's combined data would exceed the bounds of the `data` array it is copied into. This flaw can lead to out-of-bounds writing, which could corrupt memory and disrupt the program's execution or cause it to crash, potentially being exploited to induce a denial-of-service (DoS) condition. This issue emphasizes the necessity for rigorous bounds checking in functions handling external data inputs to prevent security vulnerabilities.

Relevant Code

```
// nubit-kzg/commit.go:65
func (k *KzgCommitter) Commit(leafHashes [][]byte) ([]byte, error) {
  data := gokzg4844.Blob{}
  for i, h := range leafHashes {
    copy(data[i*gokzg4844.SerializedScalarSize:
(i+1)*gokzg4844.SerializedScalarSize], h)
  }
  commit, err := k.ctx.BlobToKZGCommitment(&data, 0)
  if err != nil {
    return nil, err
  }
  return commit[:], nil
}
```

Proof of Concept (Exploit)

```go
func TestCrashKzgCommiterCommit(t *testing.T) {
  hash := sha256.New()
  commiter := NewKzgCommitter(hash)
  leafHashes := make([][]byte, 32*4096+1)
  commiter.Commit(leafHashes)
}

func TestCrashKzgCommitterOpen1(t *testing.T) {
  hash := sha256.New()
  commiter := NewKzgCommitter(hash)
  leafHashes := make([][]byte, 32*4096+1)
  commiter.Open(leafHashes, 0)
}
```

Recommended Fix

Properly handle length validation at nubit-kzg/commit.go:65.

[HIGH] Re-verify encoding of shares in nubit-rsmt2d

Severity	Category	Code Location	Status	Fix Commit
HIGH	LOGIC	rsmt2d/extendeddatacrossw..	Fixed	9b4f7d92

In the `rsmt2d` project's `extendeddatacrossword.go` file, the current implementation lacks a re-verification process for the encoding of shares when a row or column is fully recomputed from orthogonal axes. The absence of this verification after re-computation can lead to incorrect data being accepted as valid, potentially undermining the integrity of the data reconstruction process in the presence of Byzantine faults. To fortify the resilience of the system against data corruption or manipulation, it is imperative to incorporate an additional encoding verification step immediately following the verification of merkle roots. This will ensure that the data conforms to expected encoding standards post-reconstruction, thereby enhancing the overall security and reliability of the data handling process.

Relevant Code

```
  // extendeddatacrossword.go#L165-L181
  // Check that newly completed orthogonal vectors match their new merkle
roots
  for c := 0; c < int(eds.width); c++ {
    col := eds.col(uint(c))
    if col[r] != nil {
      continue // not newly completed
    }
    if noMissingData(col, r) { // completed
      err := eds.verifyAgainstColRoots(colRoots, uint(c), col, r,
rebuiltShares[c])
      if err != nil {
        var byzErr *ErrByzantineData
        if errors.As(err, &byzErr) {
          byzErr.Shares = shares
        }
        return false, false, err
      }
    }
  }
```

```
// extendeddatacrossword.go#L238-L254
  // Check that newly completed orthogonal vectors match their new merkle
roots
  for r := 0; r < int(eds.width); r++ {
    row := eds.row(uint(r))
    if row[c] != nil {
      continue // not newly completed
    }
    if noMissingData(row, c) { // completed
      err := eds.verifyAgainstRowRoots(rowRoots, uint(r), row, c,
rebuiltShares[r])
      if err != nil {
        var byzErr *ErrByzantineData
        if errors.As(err, &byzErr) {
          byzErr.Shares = shares
        }
        return false, false, err
      }
    }
  }
```

Recommended Fix

After each calling `verifyAgainstColRoots`, add an encode verification.

[MEDIUM] Deepcopy filling nil data with zeroes in nubit-rsmt2d

Severity	Category	Code Location	Status	Fix Commit
MEDIUM	LOGIC	rsmt2d/extendeddatasquare..	Fixed	9b4f7d92

In the `deepCopy` function found in the `rsmt2d` project's `extendeddatasquare.go` file, there is a flaw in handling `nil` slices. When the function encounters a `nil` value in the source array, it improperly continues to allocate a zero-length slice for that index in the destination array, rather than preserving the `nil` status. This incorrect data handling can lead to errors or inconsistencies during processes that rely on the presence of `nil` to signify uninitialized or absent data. Enhancing this function to correctly skip the copying of `nil` values would prevent such potential data integrity issues, ensuring that the `deepCopy` function faithfully replicates the structure of the source array without introducing unintended alterations.

Relevant Code

```
// extendeddatasquare.go#L271-L278
func deepCopy(original [][]byte) [][]byte {
  dest := make([][]byte, len(original))
  for i, cell := range original {
    dest[i] = make([]byte, len(cell))
    copy(dest[i], cell)
  }
  return dest
}
```

Recommended Fix

The recommended fix for this issue is to add a check for 'nil' values in the original array, and continue the loop without copying if a 'nil' value is encountered. Here's the modified code:

```
func deepCopy(original [][]byte) [][]byte {
  dest := make([][]byte, len(original))
  for i, cell := range original {
    if cell == nil {
      continue
    }
    dest[i] = make([]byte, len(cell))
    copy(dest[i], cell)
  }
  return dest
}
```

This ensures that the function does not create a slice of zero length in the destination array when it encounters a 'nil' value in the original array.

[MEDIUM] Wrong range check in ParseNamespace in nubit-validator

Severity	Category	Code Location	Status	Fix Commit
MEDIUM	LOGIC	nubit-validator/pf/pf.go	Fixed	95c725c8

The `ParseNamespace` function within the `nubit-validator` project contains a logical error in its range validation check. The condition incorrectly flags `endShare` equal to the length of `rawShares` as out-of-bounds. Since the `endShare` index is used non-inclusively in the Golang slice notation, it should be valid for `endShare` to equal the length of `rawShares`. This misinterpretation could lead to unnecessary errors and restrict the function's ability to parse namespaces up to the end of the shares array, potentially causing misbehavior in data parsing operations or when handling edge cases during namespace processing.

Relevant Code

```
// pf/pf.go#L293-L295
if endShare >= len(rawShares) {
  return appns.Namespace{}, fmt.Errorf("end share %d is higher than block
shares %d", endShare, len(rawShares))
}
```

Recommended Fix

```
// pf/pf.go#L293-L295
if endShare > len(rawShares) {
  return appns.Namespace{}, fmt.Errorf("end share %d is higher than block
shares %d", endShare, len(rawShares))
}
```

[MEDIUM] skipPadding does not skip multiple paddings in nubit-node

Severity	Category	Code Location	Status	Fix Commit
MEDIUM	LOGIC	nubit-node/strucs/btx	Fixed	b9c3ec6e

The `skipPadding` function in the `nubit-node` project's `strucs/btx` module is designed to bypass padding shares at the beginning of a list. However, the current implementation only checks and skips the first share, without considering the possibility of multiple consecutive padding shares. This limitation could lead to incorrect parsing of data blocks if multiple padding shares are present, as subsequent padding shares would not be skipped, potentially leading to erroneous processing of data. Enhancing the function to iteratively check and skip all consecutive padding shares at the start of the list would improve the robustness and accuracy of data parsing in scenarios involving variable amounts of padding.

Relevant Code

```
// strucs/btx/parser.go#L113-L134
// skipPadding skips first share in the range if this share is the Padding
share.
func (p *parser) skipPadding(shares []shares.Share) ([]shares.Share, error) {
  if len(shares) == 0 {
    return nil, errEmptyShares
  }

  isPadding, err := shares[0].IsPadding()
  if err != nil {
    return nil, err
  }

  if !isPadding {
    return shares, nil
  }

  // update blob index if we are going to skip one share
  p.index++
  if len(shares) > 1 {
    return shares[1:], nil
  }
  return nil, nil
}
```

[MEDIUM] Inconsistencies in Node States Arising from Non-Deterministic Network Requests in nubit-validator

Severity	Category	Code Location	Status	Fix Commit
MEDIUM	CONS	nubit-validator/tx/prepar..	Fixed	b8577116

`FetchHeight` uses RPC to retrieve the block height. Due to the distributed nature of the nodes, network issues might cause what should be a normal response to return an error instead. To implement this RPC call, it is necessary to consider the determinism of the results across different nodes.
tx/prepare_proposal.go riemaLabs/nubit-validator, `CheckTxsWithBtcRef` utilizes `FetchHeight` , which is used in `ProcessProposal`, which is part of the Nubit State-machine Scope. Non-determinism in the executed code can trigger the chain to fork, or cause honest validators to be unfairly penalized.

Relevant Code

```
// nubit-validator/tx/prepare_proposal.go#L76
btcHeight, err := FetchHeight(app.BTCRpc)
```

Recommended Fix

Consider implementing request retries when FetchHeight returns an error, to mitigate the uncertainty introduced by its RPC calls.

[LOW] Proper handlings of err in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
LOW	FORMAT	nubit-kzg/commit.go	Ack	N/A

The codebase of the `nubit-kzg` project exhibits several instances where errors from cryptographic operations are not adequately checked. Functions like `SetBytesCanonical` and `Write` operate on critical data and yet fail to handle possible errors that might arise during these operations. Although it's unlikely to trigger such err/panic, as a cryptographic package that takes interface, it's recommended to add error handling in them.

Relevant Code

```
// commit.go#L142
claimedValue.SetBytesCanonical(hashLeaf[:])
// commit.go#L193
sc.SetBytes(hData)
// commit.go#L224
sc.SetBytes(hData)
// commit.go#L189
h.Write(ndata)
```

[LOW] Potential integer overflow in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
LOW	FORMAT	nubit-kzg/hasher.go	Ack	N/A

After fixing the out-of-bound write in this function, one should also check if the length is greater than `gokzg4844.ScalarsPerBlob*gokzg4844.SerializedScalarSize`

Relevant Code

```
// hasher.go#L174-L176
func (e *KzgHasher) BlockSize() int {
  return gokzg4844.ScalarsPerBlob*gokzg4844.SerializedScalarSize - e.length
}
```

`e.length` is set at

```
// hasher.go#L147-L154
func (e *KzgHasher) Write(leafHash []byte) (n int, err error) {
  if len(leafHash)%gokzg4844.SerializedScalarSize != 0 {
    return 0, fmt.Errorf("leaf hash length is not a multiple of %d",
gokzg4844.SerializedScalarSize)
  }
  copy(e.data[e.length:len(leafHash)], leafHash)
  e.length = len(leafHash)
  return len(leafHash), nil
}
```

[LOW] Proper handlings of err in nubit-rsmt2d

Severity	Category	Code Location	Status	Fix Commit
LOW	FORMAT	rsmt2d/kzg.go	Ack	N/A

The codebase of the `rsmt2d` project exhibits several instances where errors from certain operations are not adequately checked. Functions like `SetBytes` and `Write` operate on critical data and yet fail to handle possible errors that might arise during these operations. Although it's unlikely to trigger such err/panic, it's recommended to add error handling in them.

Relevant Code

```
// kzg.go#L49
hasher.Write(l)
// kzg.go#L53
sc.SetBytes(h)
```

[LOW] Add index range check in QueryTxInclusionProof in nubit-validator

Severity	Category	Code Location	Status	Fix Commit
LOW	FORMAT	nubit-validator/pf/pf.go	Ack	N/A

In the `QueryTxInclusionProof` function in the `nubit-validator` project, there is no index range check for the input `index`. An index that exceeds the length of the `txs` array could lead to unexpected behavior or errors. Adding a range check for `index` to ensure it lies within the bounds of the `txs` array would increase the robustness of the function and prevent potential issues.

Relevant Code

```
// pf/pf.go#L190-L193
  index, err := strconv.ParseInt(path[0], 10, 64)
  if err != nil {
    return nil, err
  }
```

Recommended fix

```
// pf/pf.go#L190-L193
  index, err := strconv.ParseInt(path[0], 10, 64)
  if err != nil {
    return nil, err
  }

  if index < 0 {
    return nil, // Some error
  }
```

[INFO] Typos in nubit-kzg

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-kzg/commit.go	Ack	N/A

In the `nubit-kzg` project's `commit.go` file, there are several instances of typographical errors in the comments and code. While these do not affect the functionality of the code, they can potentially lead to confusion when reading or interpreting the comments. It is recommended to review and correct these for improved clarity and readability.

Relevant Code

```
// commit.go#L15
MaxDataLengh   = gokzg4844.ScalarsPerBlob
// commit.go#L110
func (k *KzgCommitter) CommentSize() int {
// kzgproof.go#L300
func (proof NamespaceRangeProof) VerifyAbsance(h hash.Hash, nID namespace.ID,
commitment []byte) bool {
```

[INFO] Return err instead of panic in nubit-kzg committer

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-kzg/commit.go	Ack	N/A

In the `nubit-kzg` project's commiter interface and implementations, there are instances where the code uses `panic` instead of returning errors. While `panic` can be useful in development, it's generally considered better practice to return errors in a production environment, allowing the calling code to decide how to handle the error. It is recommended to replace these `panic` calls with error returns to improve the robustness and error-handling capabilities of the code.

Relevant Code

```
// commit.go#L154-L157
func (n *KzgCommitter) Write(data []byte) (int, error) {
  if n.data != nil {
    panic("only a single Write is allowed")
  }
  ...
```

[INFO] Return err instead of panic in nubit-kzg hasher

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-kzg/hasher.go	Ack	N/A

In the `nubit-kzg` project's hasher interface and implementations, there are instances where the code uses `panic` instead of returning errors. While `panic` can be useful in development, it's generally considered better practice to return errors in a production environment, allowing the calling code to decide how to handle the error. It is recommended to replace these `panic` calls with error returns to improve the robustness and error-handling capabilities of the code.

Relevant Code

```
// hasher.go#L156-L160
func (e *KzgHasher) Sum(b []byte) []byte {
  c, err := e.ctx.BlobToKZGCommitment(&e.data, 0)
  if err != nil {
    panic(err)
  }
  ...
```

[INFO] Return Error for `OnRecvPacket` in nubit-validator

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-validator/utils/tk/..	Ack	N/A

Part of the TokenFilter module involves attempting to unmarshal the incoming `FungibleTokenPacketData` (https://github.com/RiemaLabs/nubit-validator/blob/main/utils/tk/tokenfilter/ibc_middleware.go#L44). If an error arises, it passes the data down to the transfer module. Currently, the transfer module only supports this packet type and will naturally produce an error as expected. However, identifying and returning the error earlier may be a more efficient approach.

Relevant Code

```
// nubit-validator/utils/tk/tokenfilter/ibc_middleware.go
func (m *tokenFilterMiddleware) OnRecvPacket(
  ctx sdk.Context,
  packet channeltypes.Packet,
  relayer sdk.AccAddress,
) exported.Acknowledgement {
  var data transfertypes.FungibleTokenPacketData
  if err := transfertypes.ModuleCdc.UnmarshalJSON(packet.GetData(), &data);
err != nil {
    // If this happens either a) a user has crafted an invalid packet, b) a
    // software developer has connected the middleware to a stack that does
    // not have a transfer module, or c) the transfer module has been modified
    // to accept other Packets. The best thing we can do here is pass the
packet
    // on down the stack.
    return m.IBCModule.OnRecvPacket(ctx, packet, relayer)
  }
```

Recommended Fix

```
if err := transfertypes.ModuleCdc.UnmarshalJSON(packet.GetData(), &data); err
!= nil {
        return channeltypes.NewErrorAcknowledgement(Err)
}
```

[INFO] DNS TTL not respected in nubit-core

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-validator/p2p/peer.go	Ack	N/A

Nodes in `nubit-core` resolve DNS entries only once at startup and continue using the same IP addresses without respecting the DNS Time-To-Live (TTL). This can lead to nodes connecting to outdated or incorrect IPs, resulting in connectivity issues or potential security risks.

Recommended Fix

Implement a mechanism to re-query DNS entries after their TTL expires to ensure nodes maintain up-to-date IP addresses.

[INFO] Add check to return early in RowProof.Validate in nubit-core

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-core/types/row_proof.go	Ack	N/A

In the `RowProof.Validate` method, there is no initial check for the condition where `rp.EndRow` is less than `rp.StartRow`. While later code may produce errors if this condition is false, the resulting error messages are not descriptive of the actual issue.

Recommended Fix

Add an early check in `RowProof.Validate` to verify if `rp.EndRow < rp.StartRow` and return a clear error message if the condition is met.

[INFO] Node version cannot be updated at runtime in nubit-core

Severity	Category	Code Location	Status	Fix Commit
INFO	FORMAT	nubit-core/rpc/core/status.go	Ack	N/A

The node's version in `nubit-core` is fixed at startup and does not update if the node's version changes at runtime without restarting. This can lead to discrepancies in version reporting, affecting status information and compatibility checks.

```
// rpc/core/status.go
  result := &ctypes.ResultStatus{
    NodeInfo: env.P2PTransport.NodeInfo().(p2p.DefaultNodeInfo),
// ...
```

Recommended Fix

Introduce a runtime method to retrieve the current binary version, ensuring that the node reports accurate version information even after updates.

Centralization Risk

No centralization risks identified at present.