CERTIK

Gonka - Ethereum Bridge

Security Assessment

CertiK Assessed on Jun 18th, 2026

CertiK Assessed on Jun 18th, 2026

Gonka - Ethereum Bridge

The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Bridge	Cosmos (ATOM) | Ethereum (ETH)	Manual Review, Static Analysis

LANGUAGE	TIMELINE
Go, Rust, Solidity	Preliminary comments published on 02/13/2026 Final report published on 06/18/2026

Vulnerability Summary

58	52	1	0	5	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

0	Centralization		Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
4	Major	4 Resolved	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
11	Medium	10 Resolved, 1 Mitigated	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
30	Minor	30 Resolved	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
13	Informational	8 Resolved, 5 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | GONKA - ETHEREUM BRIDGE

CODEBASE | GONKA - ETHEREUM BRIDGE

Repository

https://github.com/gonka-ai/gonka

Commit

82c43a42c3c2f49b56ee8a32e6458480daf39ca9

Audit Scope

The files in scope are listed in the appendix.

APPROACH & METHODS | GONKA - ETHEREUM BRIDGE

This audit was conducted for Gonka to evaluate the security and correctness of the smart contracts associated with the Gonka - Ethereum Bridge project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | GONKA - ETHEREUM BRIDGE

Overview

The **Gonka-Ethereum bridge** enables **bidirectional** cross-chain token transfers between the Gonka blockchain (Cosmos SDK-based) and Ethereum. It uses **BLS threshold signatures** for cryptographic security, where a distributed validator set signs bridge operations collectively.

Supported Asset Flows

#	Direction	From	To	Mechanism
1	ETH → Gonka	ERC-20/ETH	Wrapped CW20	Lock on ETH, Mint on Gonka
2	Gonka → ETH	Wrapped CW20	ERC-20/ETH	Burn on Gonka, Unlock on ETH
3	Gonka → ETH	Native GNK	WGNK (ERC-20)	Lock in Escrow, Mint WGNK
4	ETH → Gonka	WGNK (ERC-20)	Native GNK	Burn WGNK, Release from Escrow

BLS Threshold Signature System

The BLS (Boneh-Lynn-Shacham) threshold signature system used in the Gonka-Ethereum bridge consists of two main components:

1. **Distributed Key Generation (DKG)** - Creating shared BLS keys across validators
2. **Threshold Signing** - Producing signatures that require t-of-n validator participation

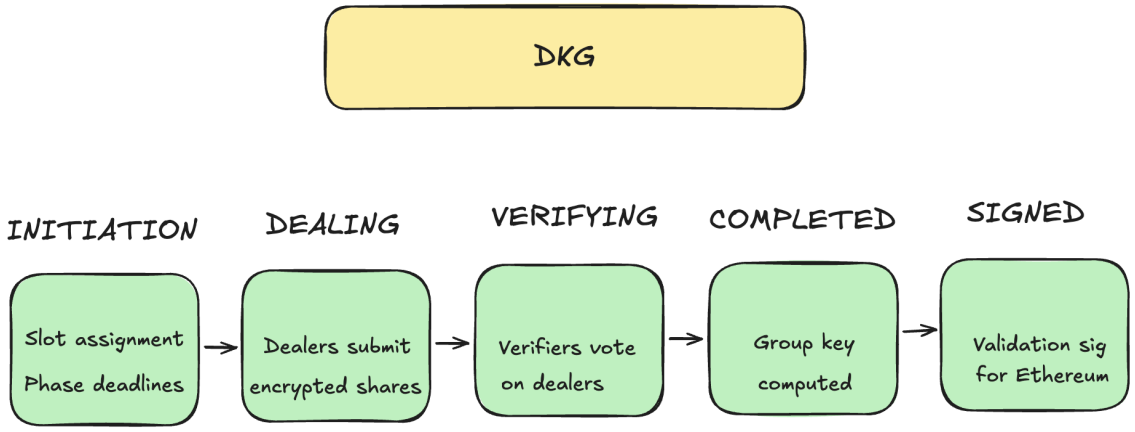
---

Part 1: Distributed Key Generation (DKG)

Overview

The DKG protocol creates a shared BLS12-381 group public key where no single validator knows the full private key. Each validator holds a share of the private key, computed from shares contributed by all dealers.

DKG Phases



Phase	Status	Description
`DKG_PHASE_DEALING`	Active	Dealers submit polynomial commitments and encrypted shares
`DKG_PHASE_VERIFYING`	Active	Verifiers decrypt shares and vote on dealer validity
`DKG_PHASE_COMPLETED`	Terminal	Group public key computed from valid dealer commitments
`DKG_PHASE_SIGNED`	Terminal	Validation signature generated for Ethereum bridge
`DKG_PHASE_FAILED`	Terminal	Insufficient participation, DKG aborted

Phase 1: Initiation

File: `inference-chain/x/bls/keeper/dkg_initiation.go`

1.1 Slot Assignment

Validators are assigned **slots** based on their staking weight. Slots represent voting power in the threshold scheme.

```
// InitiateKeyGenerationForEpoch initiates DKG for a given epoch
func (k Keeper) InitiateKeyGenerationForEpoch(ctx sdk.Context, epochID uint64,
    finalizedParticipants []types.ParticipantWithWeightAndKey) error {

    params := k.GetParams(ctx)
    iTotalSlots := params.ITotalSlots           // Total slots (e.g., 100)
    tSlotsDegree := iTotalSlots - params.TSlotsDegreeOffset  // Polynomial degree

    // Assign slots based on percentage weights
    blsParticipants, err := k.AssignSlots(ctx, finalizedParticipants, iTotalSlots)

    // Calculate phase deadlines
    dealingPhaseDeadline := currentHeight + params.DealingPhaseDurationBlocks
    verifyingPhaseDeadline := dealingPhaseDeadline +
params.VerificationPhaseDurationBlocks
}
```

Slot Assignment Algorithm (`AssignSlots`):

1. Sort participants by address (deterministic ordering)
2. Calculate `floor(weight_ratio * totalSlots)` for each participant
3. Distribute remaining slots by largest remainder
4. Ensure every participant gets at least 1 slot
5. Assign contiguous slot ranges `[SlotStartIndex, SlotEndIndex]`

Example:

```
Participant A: 40% weight → Slots [0, 39]   (40 slots)
Participant B: 35% weight → Slots [40, 74]  (35 slots)
Participant C: 25% weight → Slots [75, 99]  (25 slots)
Total: 100 slots
```

1.2 Parameters

Parameter	Description
`iTotalSlots`	Total number of slots (voting units)
`tSlotsDegree`	Polynomial degree (threshold = t+1 slots)
`DealingPhaseDurationBlocks`	Blocks allowed for dealing phase
`VerificationPhaseDurationBlocks`	Blocks allowed for verification phase

Phase 2: Dealing

Chain Files: `inference-chain/x/bls/keeper/msg_server_dealer.go` **Client Files**: `decentralized-`

`api/internal/bls/dealer.go`

Each participant acts as a **dealer**, generating and distributing secret shares.

2.1 Polynomial Generation

Each dealer `k` generates a random polynomial of degree `t`:

```
Poly_k(x) = a_{k,0} + a_{k,1}·x + a_{k,2}·x² + ... + a_{k,t}·x^t
```

Where:

- `a_{k,j}` are random scalars in the BLS12-381 scalar field `Fr`
- `a_{k,0}` is dealer k's contribution to the group secret key

```go
// generateRandomPolynomial generates random polynomial coefficients
func generateRandomPolynomial(degree uint32) ([]*fr.Element, error) {
    coefficients := make([]*fr.Element, degree+1)
    for i := uint32(0); i <= degree; i++ {
        coeff := new(fr.Element)
        coeff.SetRandom()  // Random scalar in Fr
        coefficients[i] = coeff
    }
    return coefficients, nil
}
```

2.2 Commitment Generation

Dealer computes **G2 commitments** to polynomial coefficients:

```
C_{k,j} = a_{k,j} · G2    (for j = 0 to t)
```

```go
// computeG2Commitments computes G2 commitments for polynomial coefficients
func computeG2Commitments(coefficients []*fr.Element) [][]byte {
    _, _, _, g2Gen := bls12381.Generators()  // Get G2 generator

    for i, coeff := range coefficients {
        var commitment bls12381.G2Affine
        commitment.ScalarMultiplication(&g2Gen, coeff.BigInt())
        commitments[i] = commitment.Bytes()  // 96-byte compressed G2
    }
    return commitments
}
```

2.3 Share Distribution

For each participant's slot `i`, dealer evaluates:

```
share_{k,i} = Poly_k(i+1)    // Using x = slotIndex+1 to avoid x=0
```

```
// evaluatePolynomial evaluates polynomial at given x using Horner's method
func evaluatePolynomial(polynomial []*fr.Element, x uint32) *fr.Element {
    xFr := new(fr.Element).SetUint64(uint64(x))
    result := new(fr.Element).Set(polynomial[len(polynomial)-1])

    for i := len(polynomial) - 2; i >= 0; i-- {
        result.Mul(result, xFr)
        result.Add(result, polynomial[i])
    }
    return result
}
```

2.4 Share Encryption

Shares are encrypted using **ECIES** with the recipient's secp256k1 public key:

```
// encryptForParticipant encrypts data using Cosmos-compatible ECIES
func encryptForParticipant(data []byte, secp256k1PubKeyBytes []byte) ([]byte, error)
{
    pubKey, err := secp256k1.ParsePubKey(secp256k1PubKeyBytes)
    ecdsaPubKey := pubKey.ToECDSA()
    eciesPubKey := ecies.ImportECDSAPublic(ecdsaPubKey)

    return ecies.Encrypt(rand.Reader, eciesPubKey, data, nil, nil)
}
```

2.5 Dealer Message Structure

```
message MsgSubmitDealerPart {
    string creator = 1;                                  // Dealer address
    uint64 epoch_id = 2;                                 // Target epoch
    repeated bytes commitments = 3;                      // C_{k,j} G2 points (96
bytes each)
    repeated EncryptedSharesForParticipant shares = 4;     // Encrypted shares per
participant
}

message EncryptedSharesForParticipant {
    repeated bytes encrypted_shares = 1;  // ECIES ciphertexts (one per slot × keys)
}
```

2.6 Chain Validation

```
func (ms msgServer) SubmitDealerPart(goCtx context.Context, msg
*types.MsgSubmitDealerPart) {
    // 1. Verify epoch exists and is in DEALING phase
    // 2. Check dealing phase deadline hasn't passed
    // 3. Verify creator is a participant
    // 4. Check participant hasn't already submitted
    // 5. Validate encrypted shares count matches participant count
    // 6. Store dealer part at participant's index
}
```

Phase 3: Verification

Chain Files: `inference-chain/x/bls/keeper/msg_server_verifier.go` **Client Files**: `decentralized-api/internal/bls/verifier.go`

Participants verify received shares against public commitments.

3.1 Phase Transition

Triggered when `DealingPhaseDeadlineBlock` is reached:

```
func (k Keeper) TransitionToVerifyingPhase(ctx sdk.Context, epochBLSData
*types.EpochBLSData) error {
    // Calculate slots with dealer parts
    slotsWithDealerParts := k.CalculateSlotsWithDealerParts(epochBLSData)

    // Require >50% participation
    if slotsWithDealerParts > epochBLSData.ITotalSlots/2 {
        epochBLSData.DkgPhase = types.DKGPhase_DKG_PHASE_VERIFYING
    } else {
        epochBLSData.DkgPhase = types.DKGPhase_DKG_PHASE_FAILED
    }
}
```

3.2 Share Decryption

Verifiers decrypt their shares using their Cosmos keyring:

```
func (bm *BlsManager) decryptShare(encryptedShare []byte) (*fr.Element, error) {
    // Use Cosmos keyring for decryption
    decryptedBytes, err := bm.cosmosClient.DecryptBytes(encryptedShare)

    share := &fr.Element{}
    share.SetBytes(decryptedBytes)
    return share, nil
}
```

3.3 Share Verification

Each verifier checks if decrypted share matches commitment polynomial:

```
Verify: share_{k,i} · G2 == Σ_{j=0}^{t} C_{k,j} · (i+1)^j
```

```
func (bm *BlsManager) verifyShareAgainstCommitments(share *fr.Element, slotIndex
uint32,
    commitments [][]byte) (bool, error) {

    // Evaluate commitment polynomial at slotIndex+1
    slotIndexFr := new(fr.Element).SetUint64(uint64(slotIndex + 1))
    var expectedCommitment bls12381.G2Affine
    slotIndexPower := new(fr.Element).SetOne()

    for j, commitmentBytes := range commitments {
        var commitment bls12381.G2Affine
        commitment.Unmarshal(commitmentBytes)

        var scaledCommitment bls12381.G2Affine
        scaledCommitment.ScalarMultiplication(&commitment, slotIndexPower.BigInt())
        expectedCommitment.Add(&expectedCommitment, &scaledCommitment)

        slotIndexPower.Mul(slotIndexPower, slotIndexFr)
    }

    // Compute actual commitment: share · G2
    var actualCommitment bls12381.G2Affine
    _, _, _, g2Gen := bls12381.Generators()
    actualCommitment.ScalarMultiplication(&g2Gen, share.BigInt())

    return actualCommitment.Equal(&expectedCommitment), nil
}
```

3.4 Verification Vector Submission

```
message MsgSubmitVerificationVector {
    string creator = 1;
    uint64 epoch_id = 2;
    repeated bool dealer_validity = 3;  // True/False for each dealer
}
```

3.5 Share Aggregation (Per-Slot)

Each verifier aggregates valid shares per slot:

```
// Aggregate shares from all valid dealers for each slot
for slotOffset := 0; slotOffset < numSlots; slotOffset++ {
    aggregatedShare := &fr.Element{}
    aggregatedShare.SetZero()

    for dealerIndex := range dealerParts {
        if dealerValidity[dealerIndex] {
            aggregatedShare.Add(aggregatedShare, &dealerShares[dealerIndex]
[slotOffset])
        }
    }
    verificationResult.AggregatedShares[slotOffset] = *aggregatedShare
}
```

Phase 4: Completion

File: `inference-chain/x/bls/keeper/phase_transitions.go`

4.1 Dealer Consensus

Determine valid dealers via majority voting:

```
func (k Keeper) DetermineValidDealersWithConsensus(epochBLSData *types.EpochBLSData)
([]bool, error) {
    validDealers := make([]bool, participantCount)

    for dealerIndex := 0; dealerIndex < participantCount; dealerIndex++ {
        validVotes := 0
        totalVotes := 0

        for _, verification := range epochBLSData.VerificationSubmissions {
            if verification.DealerValidity[dealerIndex] {
                validVotes++
            }
            totalVotes++
        }

        // Dealer valid if >50% approve AND submitted parts
        validDealers[dealerIndex] = (validVotes > totalVotes/2) &&
dealerSubmittedParts
    }
    return validDealers, nil
}
```

4.2 Group Public Key Computation

Aggregate `C_{k,0}` commitments from all valid dealers:

```
func (k Keeper) ComputeGroupPublicKey(epochBLSData *types.EpochBLSData, validDealers
[]bool) ([]byte, error) {
    var groupPublicKey bls12381.G2Affine  // Initialize as identity

    for dealerIndex, dealerIsValid := range validDealers {
        if !dealerIsValid {
            continue
        }

        // Get C_{k,0} (first commitment = dealer's public key contribution)
        commitmentBytes := epochBLSData.DealerParts[dealerIndex].Commitments[0]

        var commitment bls12381.G2Affine
        commitment.Unmarshal(commitmentBytes)

        // GroupPubKey = Σ C_{k,0}
        groupPublicKey.Add(&groupPublicKey, &commitment)
    }

    return groupPublicKey.Bytes(), nil  // 96-byte compressed G2
}
```

Mathematical Relationship:

```
GroupPublicKey = Σ_{k∈ValidDealers} C_{k,0}
               = Σ_{k∈ValidDealers} a_{k,0} · G2
               = (Σ_{k} a_{k,0}) · G2
               = GroupSecretKey · G2
```

Phase 5: Group Key Validation (Cross-Chain)

Chain Files: `inference-chain/x/bls/keeper/msg_server_group_validation.go` **Client Files**: `decentralized-api/internal/bls/group_validation.go`

Creates a threshold signature for Ethereum to verify the new group key.

5.1 Validation Message

Previous epoch validators sign a message containing the new group key:

```
// Message format for Ethereum verification
func computeValidationMessageHash(groupPublicKey []byte, previousEpochID uint64,
chainID string) []byte {
    var encodedData []byte

    // 1. Previous epoch ID (8 bytes, big-endian)
    previousEpochBytes := make([]byte, 8)
    binary.BigEndian.PutUint64(previousEpochBytes, previousEpochID)
    encodedData = append(encodedData, previousEpochBytes...)

    // 2. Chain ID hash (32 bytes)
    chainIdHash := sha256.Sum256([]byte(chainID))
    encodedData = append(encodedData, chainIdHash[:]...)

    // 3. Uncompressed G2 (256 bytes: X.c0||X.c1||Y.c0||Y.c1, each 64 bytes)
    // Decompress 96-byte G2 and expand to 256-byte uncompressed format

    // 4. Keccak256 hash
    hash := sha3.NewLegacyKeccak256()
    hash.Write(encodedData)
    return hash.Sum(nil)
}
```

5.2 Signature Collection

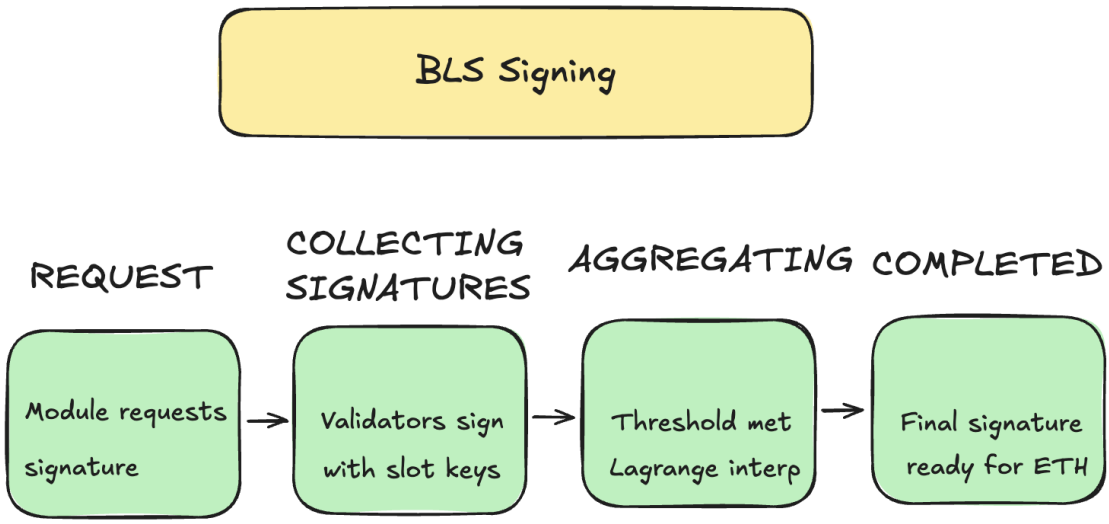
```
func (ms msgServer) SubmitGroupKeyValidationSignature(msg
*types.MsgSubmitGroupKeyValidationSignature) {
    // 1. Verify submitter was participant in previous epoch
    // 2. Verify slot ownership
    // 3. Verify BLS partial signature
    // 4. Add to partial signatures
    // 5. Check threshold (>50% of previous epoch slots)
    // 6. If threshold met: aggregate and verify final signature
    // 7. Transition to DKG_PHASE_SIGNED
}
```

Part 2: Threshold Signing

Overview

Once DKG completes, the system can produce threshold BLS signatures. Any operation requiring a threshold signature (bridge withdrawals, mints, etc.) follows this flow.

Signing Flow



Step 1: Signature Request

File: `inference-chain/x/bls/keeper/threshold_signing.go`

Modules (bridge, inference) request threshold signatures:

```
func (k Keeper) RequestThresholdSignature(ctx sdk.Context, signingData
types.SigningData) error {
    // 1. Validate current epoch completed DKG
    // 2. Validate request_id uniqueness

    // 3. Encode data using Ethereum-compatible abi.encodePacked
    encodedData := k.encodeSigningData(signingData)

    // 4. Compute message hash (keccak256)
    hash := sha3.NewLegacyKeccak256()
    hash.Write(encodedData)
    messageHash := hash.Sum(nil)

    // 5. Create and store request
    request := &types.ThresholdSigningRequest{
        RequestId:          signingData.RequestId,
        CurrentEpochId:     signingData.CurrentEpochId,
        MessageHash:        messageHash,
        Status:             types.COLLECTING_SIGNATURES,
        DeadlineBlockHeight: ctx.BlockHeight() + params.SigningDeadlineBlocks,
    }

    // 6. Emit event for validators
    ctx.EventManager().EmitTypedEvent(&types.EventThresholdSigningRequested{...})
}
```

1.1 SigningData Structure

```
type SigningData struct {
    CurrentEpochId uint64    // Epoch for signing
    ChainId        []byte    // 32-byte chain ID hash
    RequestId      []byte    // 32-byte unique request ID
    Data           [][]byte  // Additional data fields
}
```

1.2 Message Encoding (Ethereum-Compatible)

```
func (k Keeper) encodeSigningData(signingData types.SigningData) []byte {
    var encoded []byte

    // epochId (8 bytes, big-endian)
    epochBytes := make([]byte, 8)
    binary.BigEndian.PutUint64(epochBytes, signingData.CurrentEpochId)
    encoded = append(encoded, epochBytes...)

    // chainId (32 bytes)
    encoded = append(encoded, signingData.ChainId...)

    // requestId (32 bytes)
    encoded = append(encoded, signingData.RequestId...)

    // data elements (variable)
    for _, dataElement := range signingData.Data {
        encoded = append(encoded, dataElement...)
    }

    return encoded
}
```

Step 2: Partial Signature Generation

File: `decentralized-api/internal/bls/threshold_signing.go`

Validators listen for `EventThresholdSigningRequested` and sign:

```
func (bm *BlsManager) ProcessThresholdSigningRequested(event
*chainevents.JSONRPCResponse) error {
    // 1. Extract request data from event
    requestIdBytes, _ := bm.extractEventData(event, "request_id")
    messageHashBytes, _ := bm.extractEventData(event, "message_hash")
    epochId, _ := strconv.ParseUint(epochIdStr, 10, 64)

    // 2. Get cached verification result (slot shares from DKG)
    result, _ := bm.GetOrRecoverVerificationResult(epochId)

    // 3. Check if we're a participant
    if !result.IsParticipant {
        return nil  // Skip if not participating
    }

    // 4. Compute and submit partial signatures
    return bm.submitPartialSignatures(epochId, requestIdBytes, messageHashBytes,
result)
}
```

2.1 Partial Signature Computation

For each slot in the validator's range:

```
func (bm *BlsManager) computePartialSignature(messageHash []byte, result
*VerificationResult) ([]byte, error) {
    // 1. Hash message to G1 point
    messageG1, _ := bm.hashToG1(messageHash)

    // 2. For each slot, compute: signature_i = sk_i · H(m)
    var concatenated []byte
    for rel := 0; rel < len(result.AggregatedShares); rel++ {
        sk := result.AggregatedShares[rel]  // Slot's aggregated secret key share

        var sig bls12381.G1Affine
        sig.ScalarMultiplication(&messageG1, sk.BigInt())

        sigBytes := sig.Bytes()  // 48-byte compressed G1
        concatenated = append(concatenated, sigBytes[:]...)
    }
    return concatenated, nil
}
```

2.2 Hash-to-G1 (EIP-2537 Compatible)

```
func (bm *BlsManager) hashToG1(hash []byte) (bls12381.G1Affine, error) {
    // 1. Left-pad 32-byte hash to 48-byte fp.Element
    var be [48]byte
    copy(be[48-32:], hash)

    var u fp.Element
    u.SetBytes(be[:])

    // 2. SWU map to curve
    p := bls12381.MapToCurve1(&u)

    // 3. Apply isogeny
    hash_to_curve.G1Isogeny(&p.X, &p.Y)

    // 4. Clear cofactor
    var out bls12381.G1Affine
    out.ClearCofactor(&p)

    return out, nil
}
```

Step 3: Signature Submission

File: `inference-chain/x/bls/keeper/msg_server_threshold_signing.go`

```
func (ms msgServer) SubmitPartialSignature(ctx context.Context, msg
*types.MsgSubmitPartialSignature) {
    err := ms.AddPartialSignature(sdkCtx, msg.RequestId, msg.SlotIndices,
msg.PartialSignature, msg.Creator)
}
```

3.1 Validation and Storage

```
func (k Keeper) AddPartialSignature(ctx sdk.Context, requestID []byte, slotIndices
[]uint32,
    partialSignature []byte, submitter string) error {

    // 1. Get and validate request
    request, _ := k.GetSigningStatus(ctx, requestID)
    if request.Status != COLLECTING_SIGNATURES {
        return fmt.Errorf("not collecting signatures")
    }

    // 2. Check deadline
    if ctx.BlockHeight() > request.DeadlineBlockHeight {
        request.Status = EXPIRED
        return k.emitThresholdSigningFailed(ctx, requestID, epochID, "expired")
    }

    // 3. Validate slot ownership
    k.validateSlotOwnership(ctx, submitter, slotIndices, epochBLSData)

    // 4. Verify partial signature cryptographically
    k.verifyPartialSignature(partialSignature, request.MessageHash, slotIndices,
epochBLSData)

    // 5. Check for duplicate submission
    for _, existingSig := range request.PartialSignatures {
        if existingSig.ParticipantAddress == submitter {
            return fmt.Errorf("already submitted")
        }
    }

    // 6. Add partial signature
    request.PartialSignatures = append(request.PartialSignatures,
types.PartialSignature{
        ParticipantAddress: submitter,
        SlotIndices:        slotIndices,
        Signature:          partialSignature,
    })

    // 7. Check threshold and aggregate if met
    return k.checkThresholdAndAggregate(ctx, request, epochBLSData)
}
```

3.2 Partial Signature Verification

File: `inference-chain/x/bls/keeper/bls_crypto.go`

```
func (k Keeper) verifyBLSPartialSignature(signature []byte, messageHash []byte,
    epochBLSData *types.EpochBLSData, slotIndices []uint32) bool {

    // Signature is N × 48 bytes (one G1 per slot)
    sigCount := len(signature) / 48
    if sigCount != len(slotIndices) {
        return false
    }

    messageG1, _ := k.hashToG1(messageHash)
    _, _, _, g2Gen := bls12381.Generators()

    // Verify each (slot, signature) pair
    for i, slotIndex := range slotIndices {
        sigBytes := signature[i*48 : (i+1)*48]

        var g1Signature bls12381.G1Affine
        g1Signature.Unmarshal(sigBytes)

        // Compute slot public key: Σ_{k∈ValidDealers} C_k(slotIndex)
        var slotPubKey bls12381.G2Affine
        for dealerIdx, isValid := range epochBLSData.ValidDealers {
            if !isValid {
                continue
            }
            eval, _ := k.evaluateCommitmentPolynomial(dealerPart.Commitments,
slotIndex)
            slotPubKey.Add(&slotPubKey, &eval)
        }

        // Pairing check: e(sig, G2) == e(H(m), slotPubKey)
        p1, _ := bls12381.Pair([]bls12381.G1Affine{g1Signature},
[]bls12381.G2Affine{g2Gen})
        p2, _ := bls12381.Pair([]bls12381.G1Affine{messageG1},
[]bls12381.G2Affine{slotPubKey})

        if !p1.Equal(&p2) {
            return false
        }
    }
    return true
}
```

Step 4: Signature Aggregation

File: `inference-chain/x/bls/keeper/bls_crypto.go`

When threshold is met (>50% slots covered), aggregate using **Lagrange interpolation**:

```
func (k Keeper) checkThresholdAndAggregate(ctx sdk.Context, request
*types.ThresholdSigningRequest,
    epochBLSData *types.EpochBLSData) error {

    // Count covered slots
    totalSlotsCovered := uint32(0)
    for _, partialSig := range request.PartialSignatures {
        totalSlotsCovered += uint32(len(partialSig.SlotIndices))
    }

    // Threshold = >50% of total slots
    threshold := epochBLSData.ITotalSlots/2 + 1
    if totalSlotsCovered < threshold {
        return nil  // Keep collecting
    }

    // Aggregate signatures
    finalSignature, _ := k.aggregatePartialSignatures(request.PartialSignatures,
epochBLSData)

    request.Status = COMPLETED
    request.FinalSignature = finalSignature

    return k.emitThresholdSigningCompleted(ctx, request.RequestId, epochID,
finalSignature)
}
```

4.1 Lagrange Interpolation

Reconstruct signature at `x=0` from slot signatures at `x=slotIndex+1`:

```
func (k Keeper) aggregateBLSPartialSignatures(partialSignatures
[]types.PartialSignature) ([]byte, error) {
    // Flatten per-slot signatures
    var slots []uint32
    var slotSigs []slotSig  // {slot, G1 signature}

    // Compute x-coordinates for each slot (x = slotIndex + 1)
    xElems := make([]fr.Element, len(slots))
    for i, idx := range slots {
        xElems[i].SetUint64(uint64(idx + 1))
    }

    // Compute Lagrange coefficients λ_i(0)
    // λ_i(0) = Π_{j≠i} (0 - x_j) / (x_i - x_j)
    lambdaBySlot := make(map[uint32]fr.Element)
    for i := range slots {
        var numerator fr.Element
        numerator.SetOne()
        for j := range slots {
            if j == i {
                continue
            }
            var term fr.Element
            term.Neg(&xElems[j])  // 0 - x_j = -x_j
            numerator.Mul(&numerator, &term)
        }

        var denominator fr.Element
        denominator.SetOne()
        for j := range slots {
            if j == i {
                continue
            }
            var diff fr.Element
            diff.Sub(&xElems[i], &xElems[j])  // x_i - x_j
            denominator.Mul(&denominator, &diff)
        }

        var lam fr.Element
        var denInv fr.Element
        denInv.Inverse(&denominator)
        lam.Mul(&numerator, &denInv)
        lambdaBySlot[slots[i]] = lam
    }

    // Aggregate: σ = Σ λ_i · σ_i
    var aggregatedSignature bls12381.G1Affine
    aggregatedSignature.SetInfinity()
```

```
    for _, ss := range slotSigs {
        lam := lambdaBySlot[ss.slot]
        var scaledSig bls12381.G1Affine
        scaledSig.ScalarMultiplication(&ss.sig, lam.BigInt())
        aggregatedSignature.Add(&aggregatedSignature, &scaledSig)
    }

    return aggregatedSignature.Bytes(), nil  // 48-byte compressed G1
}
```

Mathematical Basis:

The secret key for slot `i` is:

```
sk_i = Σ_{k∈ValidDealers} Poly_k(i+1)
```

The partial signature is:

```
σ_i = sk_i · H(m)
```

Using Lagrange interpolation at `x=0`:

```
GroupSecretKey = Σ_i λ_i(0) · sk_i
```

Therefore:

```
FinalSignature = Σ_i λ_i(0) · σ_i
               = Σ_i λ_i(0) · sk_i · H(m)
               = GroupSecretKey · H(m)
```

Step 5: Signature Verification (On Ethereum)

File: `proposals/ethereum-bridge-contact/contracts/BridgeContract.sol`

```
function _verifyBLSSignature(
    bytes memory groupPublicKey,  // 256-byte uncompressed G2
    bytes32 messageHash,
    bytes memory signature        // 128-byte uncompressed G1
) internal view returns (bool) {
    // 1. Map message hash to G1 using EIP-2537 precompile
    bytes memory hG1 = _mapMessageToG1(messageHash);

    // 2. Negate mapped point for pairing check
    bytes memory negHG1 = _negateG1(hG1);

    // 3. Pairing check: e(sig, G2_gen) · e(-H(m), groupPubKey) == 1
    bytes memory pairingInput = abi.encodePacked(
        signature, G2_GENERATOR, negHG1, groupPublicKey
    );

    (bool success, bytes memory result) = BLS12_PAIRING.staticcall(pairingInput);
    return success && abi.decode(result, (bool));
}
```

External Dependencies

The audit scope of **Gonka - Ethereum Bridge** treats third-party entities as black boxes and assumes their functional correctness. However, in the real world, third parties can be compromised, which may lead to lost or stolen assets. There are a few dependent injection contracts or addresses in the current scope of the project:

BridgeContract

- "@openzeppelin/contracts/token/ERC20/ERC20.sol";
- "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
- "@openzeppelin/contracts/access/Ownable.sol";
- "@openzeppelin/contracts/security/ReentrancyGuard.sol";

wrapped-token

- cw20_base
- cw20
- cw2

community-sale

- cw2

The audit team assume these contracts or addresses are valid and non-vulnerable actors and implement proper logic to collaborate with the current project.

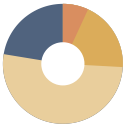
Privileged Functions

In the **Gonka - Ethereum Bridge** project, multiple privileged roles are adopted to ensure a good runtime behavior in the project, which were specified in the finding **GEB-54 | Centralization Risks in Smart Contracts**.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the vault settings and configuration according to the runtime required to best serve the community. It is also worthy of note the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

FINDINGS | GONKA - ETHEREUM BRIDGE



58	0	0	4	11	30	13
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared for Gonka to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 58 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
GEB-03	Duplicate Slot Indices Inflate Threshold Coverage	Logical Issue	Major	Resolved
GEB-12	Dealer Commitments Not Bound To Threshold Degree	Logical Issue	Major	Resolved
GEB-29	Weak Dealer Approval Enables Threshold Signing DoS	Logical Issue, Denial of Service	Major	Resolved
GEB-34	Genesis Epoch's Group Key Can Be Set By External Users	Volatile Code, Logical Issue	Major	Resolved
GEB-04	Incorrect Signing Threshold In `checkThresholdAndAggregate()`	Logical Issue	Medium	Resolved
GEB-05	Native Denom Auto-Detection Can Be Misconfigured In `community-sale` Contract	Volatile Code, Inconsistency	Medium	Resolved
GEB-13	Aggregation Of BLS Partial Signature Does Not Eliminate Duplicates	Logical Issue, Inconsistency	Medium	Resolved
GEB-14	User-Controlled RequestId Allows Front-Run Poisoning Of Threshold Signing	Logical Issue	Medium	Resolved
GEB-15	Cross-Chain Address Collision	Volatile Code, Logical Issue	Medium	Resolved
GEB-16	Bridge BLS Signatures Are Not Bound To Destination Contract	Design Issue	Medium	Resolved

ID	Title	Category	Severity	Status
GEB-17	Dealer Validation Majority Is Too Weak For Safe Key Recovery	Logical Issue	Medium	Resolved
GEB-35	Secret Shares Not Using Consensus `ValidDealers`	Coding Issue, Logical Issue	Medium	Resolved
GEB-36	Authority Mismatch In `MigrateAllWrappedTokenContracts()`	Volatile Code, Inconsistency	Medium	Resolved
GEB-54	**Operational Risks In Smart Contracts**	**Operational Risk, Centralization**	**Medium**	**Mitigated**
GEB-55	BLS Genesis Export/Import Drops In-Flight DKG And Signing State	Volatile Code, Inconsistency	Medium	Resolved
GEB-06	Known Security Issue In Upstream Dependencies	Volatile Code	Minor	Resolved
GEB-07	Weak Address Validation In `withdraw()` In `wrapped-token` Contract	Volatile Code	Minor	Resolved
GEB-08	`ADMIN` Role Cannot Not Be Updated	Inconsistency	Minor	Resolved
GEB-09	Migration From Cw20-Base Leaves Required Wrapped-Token State Uninitialized	Volatile Code, Inconsistency	Minor	Resolved
GEB-10	Migration Of `community-sale` Lacks Compatibility Checks And State Validation	Volatile Code, Inconsistency	Minor	Resolved
GEB-18	Slot Donation Picks Under-Allocated Donor, Enabling Sybil Weight Inflation	Logical Issue	Minor	Resolved
GEB-19	Secret Shares Logged In `logging.Debug`	Volatile Code	Minor	Resolved
GEB-20	Decoding Returns Zero If Fails	Volatile Code	Minor	Resolved
GEB-21	Ineffective Polynomial Degree Check In `evaluatePolynomial()`	Volatile Code	Minor	Resolved

ID	Title	Category	Severity	Status
GEB-22	Unchecked `amountToBytes32()` Panic On Oversized Amounts	Volatile Code	Minor	Resolved
GEB-23	Missing Validation Of `MsgRequestThresholdSignature.ValidateBasic()` For `chain_id` / `request_id` And Data Chunk Sizes	Volatile Code, Inconsistency	Minor	Resolved
GEB-24	Insufficient Validation For Dealer Part Submissions In `MsgSubmitDealerPart.ValidateBasic()`	Volatile Code, Inconsistency	Minor	Resolved
GEB-25	Missing Validation In Group Key Validation Signatures In `MsgSubmitGroupKeyValidationSignature.ValidateBasic()`	Volatile Code, Inconsistency	Minor	Resolved
GEB-26	Missing Validation In Partial Signature Submissions In `MsgSubmitPartialSignature.ValidateBasic()`	Volatile Code, Inconsistency	Minor	Resolved
GEB-27	Unbounded `DealerValidity` In Verification Vector Submissions Of `MsgSubmitVerificationVector.ValidateBasic()`	Volatile Code, Inconsistency	Minor	Resolved
GEB-37	DKG Process Can Be Stuck Due To Internal Errors	Logical Issue, Design Issue	Minor	Resolved
GEB-38	Inconsistent Comparison Of Deadline Block	Coding Issue	Minor	Resolved
GEB-39	Missing Validation Of `msg.Amount` Being Positive In `MsgRequestBridgeWithdrawal`	Volatile Code	Minor	Resolved
GEB-40	Broken Cleanup Logic	Logical Issue, Coding Issue	Minor	Resolved
GEB-42	Unhandled Error Of `EmitTypedEvent()`	Inconsistency	Minor	Resolved
GEB-43	Valid Dealers Can Be Less Than Threshold	Logical Issue	Minor	Resolved

ID	Title	Category	Severity	Status
GEB-47	Hard-Coded Threshold For BLS Signature	Logical Issue	Minor	Resolved
GEB-48	Missing Signed Status In `parseEpochDataFromJSON()`	Volatile Code	Minor	Resolved
GEB-49	Missing Check Of Withdrawal And Mint Amount	Volatile Code	Minor	Resolved
GEB-56	Missing Validation Of Epoch Id In `RequestThresholdSignature()`	Volatile Code, Inconsistency	Minor	Resolved
GEB-57	Slot Range Silently Clamped Instead Of Failing	Volatile Code	Minor	Resolved
GEB-59	`wrapped-token` Instances Are Not Migratable Because Deployment Leaves CW2 Marker As `crates.io:cw20-base`	Inconsistency	Minor	Resolved
GEB-60	`WGNKBurned` Can Be Emitted During `ADMIN_CONTROL`	Inconsistency	Minor	Resolved
GEB-61	Uncapped Warm-Key Fanout Can Make Dealer Parts Unsendable And Stall DKG	Volatile Code	Minor	Resolved
GEB-62	Candidate Dealer Selection Occurs Before Complaint Adjudication	Inconsistency, Logical Issue	Minor	Resolved
GEB-01	Discussion On Unpaired Burn/BurnFrom On Wrapped Supply In `wrapped-token` Contract	Inconsistency	Informational	Acknowledged
GEB-02	Discussion On `UpdateMetadata` That Modifies Decimals	Inconsistency	Informational	Acknowledged
GEB-11	`InstantiateMsg.marketing` Is Ignored	Inconsistency	Informational	Resolved
GEB-28	Discussion On Donor Mechanism	Design Issue	Informational	Resolved
GEB-33	Discussion On Old Epoch Keys Can Authorize Mint/Withdraw	Design Issue	Informational	Acknowledged

ID	Title	Category	Severity	Status
GEB-44	Discussion On DKg's Status On COMPLETED And SIGNED	Inconsistency	Informational	Resolved
GEB-45	Discussion On Slot Allocation When Participants Is More Than Slots	Logical Issue	Informational	Resolved
GEB-46	Discussion On Potential ETH/WGNK Address Collision	Logical Issue	Informational	Resolved
GEB-50	Unused Function `computeParticipantPublicKey()` In `bls_crypto.go`	Coding Style, Volatile Code	Informational	Resolved
GEB-51	Discussion On Post-Processing Of Failed Threshold Signing Requests	Design Issue	Informational	Resolved
GEB-52	Discussion On EVM Monitor And Transaction Submission	Design Issue	Informational	Acknowledged
GEB-58	`ProcessThresholdSigningRequested()` Incorrectly Returns Error	Coding Issue	Informational	Resolved
GEB-63	Discussion On Missing Emergency Pause Across Bridge Execution Paths	Volatile Code	Informational	Acknowledged

GEB-03 | Duplicate Slot Indices Inflate Threshold Coverage

Category	Severity	Location	Status
Logical Issue	Major	inference-chain/x/bls/keeper/threshold_signing.go (82c43a4): 290~293	Resolved

Description

Threshold signing accepts duplicate slot indices in a single submission. Range checks and per-slot signature checks don't dedupe, so one signer can repeat the same slot to trip the threshold early.

`inference-chain/x/bls/keeper/threshold_signing.go` , `checkThresholdAndAggregate()`

```
288 func (k Keeper) checkThresholdAndAggregate(ctx sdk.Context, request *types.
ThresholdSigningRequest, epochBLSData *types.EpochBLSData) error {
289     // Calculate total slots covered by partial signatures
290     totalSlotsCovered := uint32(0)
291     for _, partialSig := range request.PartialSignatures {
292 @>          totalSlotsCovered += uint32(len(partialSig.SlotIndices))
293     }
294
295     // Get total slots from epoch (threshold = more than 50% of slots)
296     totalSlots := epochBLSData.ITotalSlots
297     threshold := totalSlots/2 + 1
298
299     if totalSlotsCovered < threshold {
300         // Not enough signatures yet, keep collecting
301         return nil
302     }
303
304     // Threshold reached - aggregate signatures
305     finalSignature, err := k.aggregatePartialSignatures(request.
PartialSignatures, epochBLSData)
306     if err != nil {
307         // Aggregation failed - mark as failed
308         request.Status = types.
ThresholdSigningStatus_THRESHOLD_SIGNING_STATUS_FAILED
309         request.FinalSignature = []byte{}
310
311
// Remove from expiration index since it's no longer collecting signatures
312         k.removeFromExpirationIndex(ctx, request.DeadlineBlockHeight, request.
RequestId)
313
314         return k.emitThresholdSigningFailed(ctx, request.RequestId, request.
CurrentEpochId,
315             fmt.Sprintf("signature aggregation failed: %v", err))
316     }
317
318     // Success - update request with final signature
319     request.Status = types.
ThresholdSigningStatus_THRESHOLD_SIGNING_STATUS_COMPLETED
320     request.FinalSignature = finalSignature
321
322     // Remove from expiration index since it's no longer collecting signatures
323     k.removeFromExpirationIndex(ctx, request.DeadlineBlockHeight, request.
RequestId)
324
325     // Emit completion event
326     return k.emitThresholdSigningCompleted(ctx, request.RequestId, request.
CurrentEpochId,
327         finalSignature, totalSlotsCovered)
328 }
```

Aggregation computes weights on unique slots but still sums all duplicates, yielding an invalid “final” signature while marking the request completed. If downstream bridge logic trusts the module’s “completed” status without re-verifying the final signature, this is a threshold bypass and can authorize cross-chain actions with <t shares. Even if downstream re-verifies, it causes a liveness/DoS by finalizing with an unusable signature.

Recommendation

Reject any submission containing duplicate slot indices. Track coverage using unique slots across all partials for both readiness and aggregation, and only mark completed when the unique-slot threshold is met.

Alleviation

[Gonka, 03/20/2026]:
The team heeded the advice and resolved the finding by rejecting duplicate slots in the commit `bd1fd5083af80e4c8b50596becd15b042c36f0d7`.

GEB-12 | Dealer Commitments Not Bound To Threshold Degree

Category	Severity	Location	Status
Logical Issue	Major	decentralized-api/internal/bls/verifier.go (82c43a4): 397~399; inference-chain/x/bls/keeper/msg_server_dealer.go (82c43a4): 60~64; inference-chain/x/bls/types/message_submit_dealer_part.go (82c43a4): 21~23	Resolved

Description

`MsgSubmitDealerPart()` only checks that commitments are non-empty, and `verifyShareAgainstCommitments()` accepts any commitment list length. There is no enforcement that the commitment count equals the configured polynomial degree (`TSlotsDegree + 1`).

A malicious dealer can submit a higher-degree commitment set and matching shares that pass off-chain verification, effectively raising the reconstruction threshold for that dealer beyond the intended `t`. This can make the dealer "valid" yet prevent later aggregation/reconstruction, leading to liveness/DoS. It also leaves the system exposed to unbounded commitment lists that increase verification cost.

inference-chain/x/bls/keeper/msg_server_dealer.go

```
12  func (ms msgServer) SubmitDealerPart(goCtx context.Context, msg *types.
MsgSubmitDealerPart) (*types.MsgSubmitDealerPartResponse, error) {
13  //..
14      dealerPart := &types.DealerPartStorage{
15          DealerAddress:     msg.Creator,
16  @>          Commitments:       msg.Commitments,
17          ParticipantShares: participantShares,
18      }
19  //...
```

decentralized-api/internal/bls/verifier.go

```
396  func (bm *BlsManager) verifyShareAgainstCommitments(share *fr.Element,
 slotIndex uint32, commitments [][]byte) (bool, error) {
397  @>  if len(commitments) == 0 {
398          return false, fmt.Errorf("no commitments provided")
399      }
400  ...
```

This means an attacker can submit commitments and shares of a high-degree polynomial, which still passes verification `g2 * S == Sum(C_j * i^j)`. Thus the later Lagrange interpolation to reconstruct final signature will produce an wrong result, causing denial of service.

Note that the `ValidateBasic()` do not enforce the validation.

inference-chain/x/bls/types/message_submit_dealer_part.go

```
21  @>  if len(m.Commitments) == 0 {
22          return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"commitments must be non-empty")
23      }
```

Recommendation

Enforce commitment length: require `len(commitments) == params.ITotalSlots - params.TSlotsDegreeOffset + 1` in verification; reject dealer parts that deviate. Optionally cap maximum length and gas/CPU use in `ValidBasic()` .

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the checks both on-chain and off-chain. Changes have been reflected in the commit `980e112c6d59327c2ce34e19792ec9448fd272ea` .

GEB-29 | Weak Dealer Approval Enables Threshold Signing DoS

Category	Severity	Location	Status
Logical Issue, Denial of Service	Major	inference-chain/x/bls/keeper/phase_transitions.go (82c43a4): 169; inference-chain/x/bls/keeper/threshold_signing.go (82c43a4): 296 ~302	Resolved

Description

Dealer validation uses an unweighted majority of submitted DealerValidity votes and does not verify per-recipient shares against commitments. A malicious dealer can send valid shares to ~50% of recipients and garbage to the rest.

The dealer (or a colluder) can vote “true,” giving itself a bare majority among submitters (>50%). The dealer is marked "valid" and included in the group key even though many recipients lack usable shares.

`inference-chain/x/bls/keeper/phase_transitions.go`

```
154  func (k Keeper) CompleteDKG(ctx sdk.Context, epochBLSData *types.EpochBLSData)
error {
155      if epochBLSData.DkgPhase != types.DKGPhase_DKG_PHASE_VERIFYING {
156          return fmt.Errorf(
"DKG for epoch %d is not in VERIFYING phase, current phase: %s", epochBLSData.
EpochId, epochBLSData.DkgPhase.String())
157      }
158
159
// Calculate total slots covered by participants who submitted verification vectors
160      slotsWithVerification := k.CalculateSlotsWithVerificationVectors(
epochBLSData)
161
162      k.Logger().Info("Checking DKG verification participation",
163          "epochId", epochBLSData.EpochId,
164          "slotsWithVerification", slotsWithVerification,
165          "totalSlots", epochBLSData.ITotalSlots,
166          "requiredSlots", epochBLSData.ITotalSlots/2)
167
168
// Check if we have sufficient verification participation (more than half the slots)
169  @>      if slotsWithVerification > epochBLSData.ITotalSlots/2 {
170
// Sufficient verification participation - compute group public key using dealer
consensus

171          validDealers, err := k.DetermineValidDealersWithConsensus(epochBLSData)
172          if err != nil {
173              return fmt.Errorf(
"failed to determine valid dealers for epoch %d: %w", epochBLSData.EpochId, err)
174          }
175
176          groupPublicKey, err := k.ComputeGroupPublicKey(epochBLSData,
 validDealers)
177          if err != nil {
178              return fmt.Errorf(
"failed to compute group public key for epoch %d: %w", epochBLSData.EpochId, err)
179          }
180
181          // Store group public key and mark as completed
182          epochBLSData.GroupPublicKey = groupPublicKey
183          epochBLSData.DkgPhase = types.DKGPhase_DKG_PHASE_COMPLETED
184
185          // Store valid dealers in epoch data
186          epochBLSData.ValidDealers = validDealers
187
188          // Store updated epoch data
189          k.SetEpochBLSData(ctx, *epochBLSData)
190
191          // Clear active epoch since DKG process is complete (successfully)
192          k.ClearActiveEpochID(ctx)
193
```

```
194         // Emit event for successful DKG completion
195         if err := ctx.EventManager().EmitTypedEvent(&types.
EventGroupPublicKeyGenerated{
196             EpochId:        epochBLSData.EpochId,
197             GroupPublicKey: groupPublicKey,
198             ITotalSlots:    epochBLSData.ITotalSlots,
199             TSlotsDegree:   epochBLSData.TSlotsDegree,
200             EpochData:      *epochBLSData,
201             ChainId:        ctx.ChainID(),
202         }); err != nil {
203             return fmt.Errorf(
"failed to emit EventGroupPublicKeyGenerated for epoch %d: %w", epochBLSData.EpochId
, err)
204         }
205 //...
```

Later, the dealer (or colluder) withholds its partial signature, pushing the usable signers below the >50% slot threshold. Threshold-signing requests then stall and expire, which leads to a sustained liveness/DoS risk, even with <50% malicious participants plus abstentions.

`inference-chain/x/bls/keeper/threshold_signing.go`

```
288 func (k Keeper) checkThresholdAndAggregate(ctx sdk.Context, request *types.
ThresholdSigningRequest, epochBLSData *types.EpochBLSData) error {
289     // Calculate total slots covered by partial signatures
290     totalSlotsCovered := uint32(0)
291     for _, partialSig := range request.PartialSignatures {
292         totalSlotsCovered += uint32(len(partialSig.SlotIndices))
293     }
294
295     // Get total slots from epoch (threshold = more than 50% of slots)
296     totalSlots := epochBLSData.ITotalSlots
297     threshold := totalSlots/2 + 1
298
299 @>    if totalSlotsCovered < threshold {
300         // Not enough signatures yet, keep collecting
301         return nil
302     }
303 //...
```

Recommendation

Short-term mitigation includes:

- ignoring dealer self-votes,
- `≥2/3` (or at least far from 50%) of total slot weight (not just submitters) to vote,
- less weight (for example, `1/2`) than that of verification threshold to approve,
- optionally treat missing/short DealerValidity as “no” so abstentions can’t shrink the denominator and consider

Long-term solution includes share correctness checks:

- before marking a dealer valid, verify each recipient's share against the commitment polynomial (or require a zk proof).
- alternatively, add a complaint phase where any bad share excludes the dealer.

Alleviation

[Gonka, 03/27/2026]:
Issue acknowledged. The team resolved the finding by using an optimistic + dispute model:

- a true vote now needs a cryptographic proof that the voter really has usable shares for its own slots
- dealer acceptance is now slot-weighted and excludes self-vote
- if someone votes false, they are expected to attach complaint evidence, and an invalid or missing dealer response removes the dealer in dispute resolution.

Changes have been reflected in the PR: https://github.com/gonka-ai/gonka/pull/825

[CertiK, 04/23/2026]:
In the commit `a1119b819e0ced3383f0b272b7c04487d24d12f5`, the changes made to `DetermineValidDealersWithConsensus()` require a strict majority of total slots (`totalSlots/2 + 1`) to approve each dealer, but it always excludes the dealer's own verification vote. As a result, any dealer controlling at least half of the total slots cannot mathematically reach quorum even if every non-self verifier approves and the dealer submitted valid parts.

This invalid result flows into `transitionFromVerifyingToDisputing()`, which sums the slots of `candidateValidDealers` and fails the epoch when they do not exceed half of total slots. In skewed slot distributions such as 50/50 or 60/40, an honest high-weight dealer can be excluded for arithmetic reasons alone, causing the DKG epoch to enter FAILED instead of progressing to DISPUTING.

Align the quorum model with self-vote exclusion. Either count the dealer's own vote when evaluating that dealer, or compute the approval threshold against the maximum available non-self voting weight rather than against all slots.

Also ensure the candidate-dealer threshold in `transitionFromVerifyingToDisputing()` remains compatible with the dealer-validation rule so honest high-weight dealers do not deterministically fail epoch progression.

[Gonka, 05/13/2026]:
Issue acknowledged. Changes have been reflected in the commit `94c6018dee45b513150de455d4de4c84292afddf`.

[CertiK, 05/15/2026]:
The current implementation materially mitigates the original finding through an optimistic-dispute model: non-self true votes require proof of usable shares, dealer approval is slot-weighted, and invalid-share recipients can submit valid complaints that remove a dealer if the dealer cannot provide a valid response.

Under normal operation, the original attack should fail because at least one bad-share slot holder is expected to complain, and a malicious dealer that sent garbage shares cannot successfully answer that complaint.

A residual liveness corner case remains: if all holders of bad shares, weighted by enough slots to matter, abstain or fail to complain before the dispute window, the dealer may still be accepted and later withhold its own partial signature. Therefore,

the finding should be considered mitigated, but not fully eliminated; this is a limitation of the current optimistic-dispute design.

GEB-34 | Genesis Epoch's Group Key Can Be Set By External Users

Category	Severity	Location	Status
Volatile Code, Logical Issue	Major	proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (82c43a4): 302~307	Resolved

Description

`submitGroupKey()` is used to submit a new group public key for the next epoch. However, if an external user submit a group key before the admin calling `setGroupKey()` on epoch 1, it can hijack the genesis epoch, thereby control the signature for bridge workflow.

proposals/ethereum-bridge-contact/contracts/BridgeContract.sol

```
287     function submitGroupKey(
288         uint64 epochId,
289         bytes calldata groupPublicKey,
290         bytes calldata validationSig
291     ) public {
292         // Verify sequential submission
293         if (epochId != epochMeta.latestEpochId + 1) {
294             revert InvalidEpochSequence();
295         }
296
297         // Verify group public key is 256 bytes (G2 point uncompressed)
298         require(groupPublicKey.length == 256, "Invalid group key length");
299
300         // Verify validation signature against previous epoch (if not genesis)
301         GroupKey memory newGroupKeyStruct = _bytesToGroupKey(groupPublicKey);
302 @>      if (epochId > 1) {
303             GroupKey memory prevGroupKeyStruct = epochGroupKeys[epochId - 1];
304             require(!_isGroupKeyEmpty(prevGroupKeyStruct),
"Previous epoch not found");
305
306             require(_verifyTransitionSignature(prevGroupKeyStruct,
 newGroupKeyStruct, validationSig, epochId - 1), "Invalid transition signature");
307         }
308    ...
```

The code will skip the signature verifying and other checks if `eopchId <= 1` .

Recommendation

Disallow the submisson of epoch 1, adding code like `require(epochId > 1, "Epoch 1 must be set via Admin")` .

Alleviation

[Gonka, 03/26/2026]:

Issue acknowledged. The team resolved the finding by add the validation to ensure `epochId > 1`. Changes have been reflected in the commit hash: `45e63592a1360f72eab1aca3ec137b5d331c69d9`.

GEB-04 Incorrect Signing Threshold In `checkThresholdAndAggregate()`

Category	Severity	Location	Status
Logical Issue	Medium	inference-chain/x/bls/keeper/threshold_signing.go (82c43a4): 295~302	Resolved

Description

The threshold calculation in `checkThresholdAndAggregate()` is incorrect. The function has hardcoded the threshold as `totalSlots/2 + 1`, but the actual mathematical threshold should be `tSlotsDegree + 1` where `tSlotsDegree := iTotalSlots - params.TSlotsDegreeOffset`.

inference-chain/x/bls/keeper/threshold_signing.go, `checkThresholdAndAggregate()`

```
290      totalSlotsCovered := uint32(0)
291      for _, partialSig := range request.PartialSignatures {
292          totalSlotsCovered += uint32(len(partialSig.SlotIndices))
293      }
294
295      // Get total slots from epoch (threshold = more than 50% of slots)
296      totalSlots := epochBLSData.ITotalSlots
297  @>  threshold := totalSlots/2 + 1
298
299      if totalSlotsCovered < threshold {
300          // Not enough signatures yet, keep collecting
301          return nil
302      }
```

If governance changes `TSlotsDegreeOffset` away from 50, signing will finalize with too few shares (bypass) or too many (permanent liveness failure). With offset lowered (e.g., t=90), the module marks requests completed at 51 slots, potentially emitting an invalid “final” signature with <t+1 shares.

Recommendation

Use the configured threshold, `threshold := epochBLSData.TSlotsDegree + 1` (or recompute from `ITotalSlots - params.TSlotsDegreeOffset`) when checking readiness in threshold signing.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by replacing the hardcoded `>50%` signing threshold with the epoch-specific DKG threshold `TSlotsDegree + 1` in both threshold-signing flows. Changes have been reflected in the commit `6b11983b8429dbb2418888cde95b147dda25eae8`.

GEB-05 Native Denom Auto-Detection Can Be Misconfigured In `community-sale` Contract

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	inference-chain/contracts/community-sale/src/contract.rs (82c43a4): 92	Resolved

Description

The `contract.rs` in community-sale sets `config.native_denom` during instantiate by calling `get_native_denom()`, which selects the first denom from bank/TotalSupply (or falls back to ngonka).

`inference-chain/contracts/community-sale/src/contract.rs`

```
147 #[entry_point]
148 pub fn instantiate(
149     deps: DepsMut,
150     _env: Env,
151     _info: MessageInfo,
152     msg: InstantiateMsg,
153 ) -> Result<Response, ContractError> {
154     set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)
155         .map_err(|e| ContractError::Std(StdError::msg(e.to_string())))?;
156
157     let admin = deps.api.addr_validate(&msg.admin)?.to_string();
158     let buyer = deps.api.addr_validate(&msg.buyer)?.to_string();
159
160     if msg.price_usd.is_zero() {
161         return Err(ContractError::ZeroAmount {});
162     }
163
164     if msg.accepted_chain_id.is_empty() || msg.accepted_eth_contract.is_empty()
 {
165         return Err(ContractError::Std(StdError::msg(
"accepted_chain_id and accepted_eth_contract required")));
166     }
167
168 @>  let native_denom = get_native_denom(deps.as_ref())?;
169
170     let config = Config {
171         admin: admin.clone(),
172         buyer: buyer.clone(),
173         accepted_chain_id: msg.accepted_chain_id.clone(),
174         accepted_eth_contract: msg.accepted_eth_contract.to_lowercase(),
175         price_usd: msg.price_usd,
176         native_denom: native_denom.clone(),
177         is_paused: false,
178         total_tokens_sold: Uint128::zero(),
179     };
180     CONFIG.save(deps.storage, &config)?;
181
182     Ok(Response::new()
183         .add_attribute("method", "instantiate")
184         .add_attribute("admin", admin)
185         .add_attribute("buyer", buyer)
186         .add_attribute("accepted_chain_id", msg.accepted_chain_id)
187         .add_attribute("accepted_eth_contract", msg.accepted_eth_contract)
188         .add_attribute("price_usd", msg.price_usd)
189         .add_attribute("native_denom", native_denom))
190 }
```

```
fn get_native_denom(deps: Deps) -> Result<String, ContractError> {
    let request = QueryTotalSupplyRequest {};
    match query_proto::<QueryTotalSupplyRequest, QueryTotalSupplyResponse>(
        deps,
        "/cosmos.bank.v1beta1.Query/TotalSupply",
        &request,
    ) {
        Ok(response) => {
@>          if let Some(coin) = response.supply.first() {
                if !coin.denom.is_empty() {
                    return Ok(coin.denom.clone());
                }
            }
            Ok("ngonka".to_string())
        }
        Err(_) => Ok("ngonka".to_string()),
    }
}
```

An unprivileged party can mint/introduce an IBC or factory denom that sorts first, causing the contract to lock onto the wrong denom permanently. All sales, withdrawals, and “emergency” withdraws then use that attacker-chosen denom, leading to purchase DoS (InsufficientBalance vs real inventory), wrong-asset payouts, and inability to recover the intended native funds or other tokens.

The attack is realistic, as the following query result shows that the first denomination is not "ngonka" on mainnet.

```
./inferenced query bank total-supply --node tcp://185.216.21.98:8000/chain-rpc/ --
output json
```

```
{
  "supply": [
    {
      "denom":
"ibc/050330FBDF981EB546270962B8F6F9658F288E21D2166DFF2ABDBA53338BC1AD",
      "amount": "100000"
    },
    {
      "denom":
"ibc/0C9E2F9CC7C99D6C274F148C313E4231F383E8AAD4E1C18A260E5479F96C62ED",
      "amount": "3092333"
    },
    {
      "denom":
"ibc/2CC0B1B7A981ACC74854717F221008484603BB8360E81B262411B0D830EDE9B0",
      "amount": "1005114"
    },
    {
      "denom":
"ibc/4F64FF736B88BA8EA521D6D515D9652B7C23C903BF91ECEFB3991888F3C023F4",
      "amount": "43093122201"
    },
    {
      "denom":
"ibc/8E27BA2D5493AF5636760E354E46004562C46AB7EC0CC4C1CA14E9E20E2545B5",
      "amount": "209365"
    },
    {
      "denom":
"ibc/B27B5CC721C3CB27B4B61BE8BEF3ECFCC3F6D9D4ED0DA8AC8634A534C54DD527",
      "amount": "1913172735"
    },
    {
      "denom":
"ibc/C0E66D1C81D8AAF0E6896E05190FDFBC222367148F86AC3EA679C28327A763CD",
      "amount": "99900"
    },
    {
      "denom":
"ibc/D3AD3CA38F2C1AB7F5406FFB5BF10701F2004B646FD806AE5767DDEC02F916F2",
      "amount": "100000"
    },
    {
      "denom": "ngonka",
      "amount": "513028118003208546"
    }
  ],
  "pagination": {
    "total": "9"
```

```
    }
}
```

Recommendation

Recommend using the hardcoded native denom, "ngonka".

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by directly passing the native denom. Changes have been reflected in the commit `8fbed2495ac6fe6b36aa234b92f24ca08243ebca` .

GEB-13 Aggregation Of BLS Partial Signature Does Not Eliminate Duplicates

Category	Severity	Location	Status
Logical Issue, Inconsistency	Medium	inference-chain/x/bls/keeper/bls_crypto.go (82c43a4): 194~198	Resolved

Description

In `aggregateBLSPartialSignatures()`, slot indices are deduplicated only for computing Lagrange coefficients, but all signatures (including duplicates) are still accumulated using the same lambda. A payload containing the same slot index twice will over-weight that slot and produce an invalid aggregate signature instead of failing early. A malicious submitter can exploit this to cause aggregation errors or signing liveness failures.

inference-chain/x/bls/keeper/bls_crypto.go

```
178     for i, ps := range partialSignatures {
179         if len(ps.Signature)%48 != 0 {
180             return nil, fmt.Errorf(
"invalid signature payload at index %d: length=%d", i, len(ps.Signature))
181         }
182         count := len(ps.Signature) / 48
183         if count != len(ps.SlotIndices) {
184             return nil, fmt.Errorf(
"signature count mismatch at index %d: sigs=%d slots=%d", i, count, len(ps.
SlotIndices))
185         }
186         for j := 0; j < count; j++ {
187             slot := ps.SlotIndices[j]
188             start := j * 48
189             end := start + 48
190             var g1 bls12381.G1Affine
191             if err := g1.Unmarshal(ps.Signature[start:end]); err != nil {
192                 return nil, fmt.Errorf(
"failed to unmarshal signature at batch %d item %d: %w", i, j, err)
193             }
194 @>          slotSigs = append(slotSigs, slotSig{slot: slot, sig: g1})
195             if _, ok := slotSeen[slot]; !ok {
196                 slotSeen[slot] = struct{}{}
197                 slots = append(slots, slot)
198             }
199         }
200     }
```

Recommendation

Enforce uniqueness of `SlotIndices` before aggregation. Reject any partial signature payload (and any aggregate set) that contains duplicate slot indices, or skip duplicates consistently so each slot is counted exactly once. Add a duplicate check in AddPartialSignature/ValidateBasic and hard-fail aggregateBLSPartialSignatures if duplicates are detected.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by rejecting duplicate slot indices during BLS partial signature aggregation in both aggregation backends, preventing duplicated slots from being over-weighted in the final aggregate signature. Changes have been reflected in the commit `dd487edfd1cb6513374a741dbf94cbf3ab246f43` .

GEB-14 User-Controlled RequestId Allows Front-Run Poisoning Of Threshold Signing

Category	Severity	Location	Status
Logical Issue	Medium	inference-chain/x/bls/keeper/threshold_signing.go (82c43a4): 33~41	Resolved

Description

`MsgRequestThresholdSignature()` is permissionless and lets users supply arbitrary `RequestId`. The keeper enforces uniqueness and rejects any existing key. An external actor can front-run a predictable RequestId (e.g., one derived from a tx hash or module payload) and prepopulate the store, causing the legitimate internal call to fail with “request_id already exists.” This enables griefing/DoS of signing requests.

inference-chain/x/bls/keeper/threshold_signing.go

```
32  // Validate uniqueness - ensure request_id doesn't already exist
33  key := types.ThresholdSigningRequestKey(signingData.RequestId)
34  kvStore := k.storeService.OpenKVStore(ctx)
35  existingValue, err := kvStore.Get(key)
36  if err != nil {
37      return fmt.Errorf("failed to check request uniqueness: %w", err)
38  }
39  if existingValue != nil {
40      return fmt.Errorf("request_id already exists: %x", signingData.RequestId)
41  }
```

A external user can create a same `RequestId` with an existed internal transaction(usually `RequestId` is the `tx_hash`), and front-run the function `RequestThresholdSignature` to poison the `kvStore`, causing the internal call to fail.

Recommendation

Make `RequestId` non-user-controlled or scoped: derive it on-chain from entropy/nonce, or namespace by creator (creator, requestId).

Alleviation

[Gonka, 03/26/2026]:
Issue acknowledged. The team resolved the finding by deriving a new ID as `keccak256(creator || request_id)`.
Changes have been reflected in the commit `c50d7b1dd125f415433ed9203f4e0c63edee10ec`.

GEB-15 | Cross-Chain Address Collision

Category	Severity	Location	Status
Volatile Code, Logical Issue	Medium	inference-chain/x/inference/keeper/bridge_native.go (82c43a4): 39 ~53; inference-chain/x/inference/keeper/msg_server_register_bridge_addresses.go (82c43a4): 24~37	Resolved

Description

`RegisterBridgeAddresses()` stores bridge contract addresses under a `(chainId,address)` key, allowing the same address to be registered for multiple EVM chains. However, downstream logic (`IsBridgeContractAddress` -> `handleCompletedBridgeTransaction`) determines whether a bridge transaction is a native-token release using only the `contractAddress` (case-insensitive) and ignores `bridgeTx.ChainId` and the stored `BridgeContractAddress.ChainId` .

inference-chain/x/inference/keeper/msg_server_register_bridge_addresses.go

```
24      if k.HasBridgeContractAddress(ctx, chainId, address) {
25          k.LogWarn("Register bridge addresses: Address already registered",
26              types.Messages,
27              "chainId", chainId,
28              "address", address)
29          continue
30      }
31
32      bridgeAddr := types.BridgeContractAddress{
33          Id:      k.generateBridgeAddressKey(ctx, chainId, address),
34          ChainId: chainId,
35          Address: address,
36      }
37  @>  k.SetBridgeContractAddress(ctx, bridgeAddr)
```

inference-chain/x/inference/keeper/bridge_native.go

```
  39    func (k Keeper) IsBridgeContractAddress(ctx context.Context, contractAddress
string) (bool, string) {
  40      // Get all registered bridge contract addresses
  41      allBridgeAddresses := k.GetAllBridgeContractAddresses(ctx)
  42
  43      // Normalize the input address for comparison
  44      normalizedInput := strings.ToLower(contractAddress)
  45
  46      for _, bridgeAddr := range allBridgeAddresses {
  47  @>          if strings.ToLower(bridgeAddr.Address) == normalizedInput {
  48              return true, bridgeAddr.ChainId
  49          }
  50      }
  51
  52      return false, ""
  53    }
```

As a result, any registered bridge contract address becomes effectively "global": a `BridgeTransaction` referencing the same contract address on a different origin chain will still be processed as a native-token release. This can lead to incorrect processing (native release vs. wrapped-token mint) and can allow escrow releases from unintended origin chains in realistic scenarios where the same EVM address can exist on multiple chains (e.g., CREATE2/deterministic deployments).

Recommendation

Recommend making native-bridge contract identification chain-aware (e.g., IsBridgeContractAddress(chainId,address) that checks the exact (chainId,address) pair, and enforce bridgeTx.ChainId == expectedChainId), and/or enforce global uniqueness of bridge addresses at registration time via a reverse index keyed by normalized address.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the reported cross-chain address collision by updating bridge contract identification to use the exact `(chainId, address)` pair instead of address-only matching. The relevant changes are included in `8fbed2495ac6fe6b36aa234b92f24ca08243ebca`.

GEB-16 | Bridge BLS Signatures Are Not Bound To Destination Contract

Category	Severity	Location	Status
Design Issue	Medium	inference-chain/x/inference/keeper/msg_server_request_bridge_mint.go (82c43a4): 171~176; inference-chain/x/inference/keeper/msg_server_request_bridge_withdrawal.go (82c43a4): 171~177	Resolved

Description

`RequestBridgeMint()` validates that the destination `chainId` has one or more registered bridge contract addresses (`GetBridgeContractAddressesByChain()`), but then **never uses any of those addresses** when constructing the BLS-signed payload.

The signed message is built by `prepareBridgeMintSignatureData(...)` as: `[ethereumChainId, MINT_OPERATION, recipient, amount]` (with the BLS system prepending `[epochId, gonkaChainId, requestId]`).

inference-chain/x/inference/keeper/msg_server_request_bridge_mint.go

```
159  func (k Keeper) prepareBridgeMintSignatureData(chainId, recipient, amount
string) [][]byte {
160
// This function only prepares the data that comes AFTER epochId, gonkaChainId, and
requestId

161
// Final message format: [epochId, gonkaChainId, requestId, ethereumChainId,
MINT_OPERATION, recipient, amount]

162
// BLS system will prepend: epochId (8 bytes) + gonkaChainId (32 bytes) + requestId
(32 bytes)

163
164      // Use helper functions for consistent encoding
165      ethereumChainIdBytes := chainIdToBytes32(chainId)
166      recipientBytes := ethereumAddressToBytes(recipient)
167      amountBytes := amountToBytes32(amount)
168
169      // Return the data fields that come after epochId, gonkaChainId, requestId
170
// Order: ethereumChainId (32 bytes) + MINT_OPERATION (32 bytes) + recipient (20
bytes) + amount (32 bytes)

171  @>  data := [][]byte{
172          ethereumChainIdBytes, // ETHEREUM_CHAIN_ID (32 bytes)
173          mintOperationHash[:], // MINT_OPERATION hash (32 bytes)
174          recipientBytes,       // Recipient address (20 bytes)
175          amountBytes,          // Amount as uint256 (32 bytes)
176      }
177
178      return data
179  }
```

Critically, the payload does not include (i.e., is not domain-separated by) the specific destination bridge contract address that is supposed to consume the signature.

As a result, the same BLS signature produced for one deposit is valid for **any** destination bridge contract instance that:

- uses the same BLS verification key/system, and
- implements the same message format.

Because replay protection on destination contracts is typically tracked per-contract (e.g., `usedRequestIds[requestId]` stored in that contract), a single `(requestId, signature)` can be accepted once per bridge contract address. This allows minting multiple times for one escrow deposit whenever more than one destination bridge contract is active/authorized on that chain (a realistic situation during upgrades, parallel deployments, or multiple gateways).

Same issue also exists in withdrawal logic.

inference-chain/x/inference/keeper/msg_server_request_bridge_withdrawal.go

```
171     data := [][]byte{
172         ethereumChainIdBytes,     // ETHEREUM_CHAIN_ID (32 bytes)
173         withdrawOperationHash[:], // WITHDRAW_OPERATION hash (32 bytes)
174         recipientBytes,           // Recipient address (20 bytes)
175         tokenBytes,               // Token contract address (20 bytes)
176         amountBytes,              // Amount as uint256 (32 bytes)
177     }
```

Recommendation

Recommend binding each mint signature to a single destination bridge contract.

Alleviation

[Gonka, 03/27/2026]:

Issue acknowledged. The team resolved the finding by adding `DestinationBridgeAddress` to the payload and relevant validations. Changes have been reflected in the commit `c6235cb9cb89354545e49ef121618604de38bb53`.

GEB-17 | Dealer Validation Majority Is Too Weak For Safe Key Recovery

Category	Severity	Location	Status
Logical Issue	Medium	inference-chain/x/bls/keeper/phase_transitions.go (82c43a4): 296	Resolved

Description

`DetermineValidDealersWithConsensus()` determines which dealers are valid based on majority consensus from verification vectors, while it marks a dealer valid when `validVotes > totalVotes/2` among only the submissions received, with equal weight per verifier. Abstentions shrink the denominator, dealer self/ally votes count, and votes are not weighted by slot share.

`inference-chain/x/bls/keeper/phase_transitions.go`, `DetermineValidDealersWithConsensus()`

```
269  func (k Keeper) DetermineValidDealersWithConsensus(epochBLSData *types.
EpochBLSData) ([]bool, error) {
270      participantCount := len(epochBLSData.Participants)
271      if participantCount == 0 {
272          return nil, fmt.Errorf("no participants found for epoch %d",
 epochBLSData.EpochId)
273      }
274
275      validDealers := make([]bool, participantCount)
276
277      // For each dealer, count verification votes
278      for dealerIndex := 0; dealerIndex < participantCount; dealerIndex++ {
279          validVotes := 0
280          totalVotes := 0
281
282          // Count votes from all verifiers who submitted verification vectors
283          for _, verification := range epochBLSData.VerificationSubmissions {
284              if verification != nil && len(verification.DealerValidity) > 0 {
285                  // Check if this verification has a vote for this dealer
286                  if dealerIndex < len(verification.DealerValidity) {
287  @>                      totalVotes++
288                      if verification.DealerValidity[dealerIndex] {
289  @>                          validVotes++
290                      }
291                  }
292              }
293          }
294
295
// Dealer is valid if more than 50% of verifiers approve AND they submitted dealer
parts

296  @>          dealerIsValid := totalVotes > 0 && validVotes > totalVotes/2
297          dealerSubmittedParts := dealerIndex < len(epochBLSData.DealerParts) &&
298              epochBLSData.DealerParts[dealerIndex] != nil &&
299              epochBLSData.DealerParts[dealerIndex].DealerAddress != ""
300
301          validDealers[dealerIndex] = dealerIsValid && dealerSubmittedParts
302      }
303
304      return validDealers, nil
305  }
```

As a result, a dealer that sent bad shares to some recipients can still be approved with a thin subset majority, leaving parts of the cohort unable to derive usable key shares and causing signing liveness failures.

Recommendation

Require a quorum and approval relative to total active participants, not just submitters; treat missing/short DealerValidity as “no”; optionally disallow dealer self-votes.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by requiring the valid votes no less than `participantCount/2 + 1`. Changes have been reflected in the commit `73c9a09d383a0683a68c67fbc273508edb2fb409`.

GEB-35 | Secret Shares Not Using Consensus `ValidDealers`

Category	Severity	Location	Status
Coding Issue, Logical Issue	Medium	decentralized-api/internal/bls/verifier.go (82c43a4): 156~160, 277~297	Resolved

Description

The function `performVerificationAndReconstruction()` is used to verify dealers' shares and reconstruct the per-slot share. However, it just uses locally computed `verificationResult.DealerValidity` to aggregate the shares, rather then the consensus `epochData.ValidDealers`.

decentralized-api/internal/bls/verifier.go, `performVerificationAndReconstruction()`

```
277      // Now aggregate shares per slot
278      for slotOffset := 0; slotOffset < numSlots; slotOffset++ {
279          slotIndex := verificationResult.SlotRange[0] + uint32(slotOffset)
280          aggregatedShare := &fr.Element{}
281          aggregatedShare.SetZero()
282
283          // Sum up shares from all valid dealers for this slot
284          for dealerIndex := 0; dealerIndex < len(dealerParts); dealerIndex++ {
285              if verificationResult.DealerValidity[dealerIndex] && len(
verificationResult.DealerShares[dealerIndex]) > slotOffset {
286                  aggregatedShare.Add(aggregatedShare, &verificationResult.
DealerShares[dealerIndex][slotOffset])
287              }
288          }
289
290          // Store aggregated share
291  @>      verificationResult.AggregatedShares[slotOffset] = *aggregatedShare
292
293          logging.Debug(verifierLogTag+"Completed slot share reconstruction",
 inferenceTypes.BLS,
294              "slotIndex", slotIndex,
295              "slotOffset", slotOffset,
296              "finalShare", aggregatedShare.String())
297      }
```

The consensus `epochData.ValidDealers` will later be stored in `verificationResult`, but the `AggregatedShares` field is never updated and be used straightly to sign the message.

```
156      if epochData.DkgPhase == types.DKGPhase_DKG_PHASE_COMPLETED ||
157          epochData.DkgPhase == types.DKGPhase_DKG_PHASE_SIGNED {
158  @>      verificationResult.ValidDealers = epochData.ValidDealers
159          verificationResult.GroupPublicKey = epochData.GroupPublicKey
160      }
```

Thus if there exits a participant whose local `DealerValidity` is different from the consensus `ValidDealers`, the signature generated by this participant will fail the validation.

In particular, it would lead to incorrect computation of `AggregatedShares` in `decentralized-api/internal/bls/verifier.go`.

decentralized-api/internal/bls/threshold_signing.go, `computePartialSignature()`

```
134      for rel := 0; rel < len(result.AggregatedShares); rel++ {
135          sk := result.AggregatedShares[rel]
136          var sig bls12381.G1Affine
137          sig.ScalarMultiplication(&messageG1, sk.BigInt(new(big.Int)))
138          sb := sig.Bytes()
139          concatenated = append(concatenated, sb[:]...)
140      }
```

The signing service uses the incorrect local `AggregatedShares`, rather than `epochData.ValidDealers`.

Recommendation

Recommend recomputing `verificationResult.AggregatedShares` after the update of consensus data `epochData.ValidDealers`.

Alleviation

[Gonka, 04/06/2026]:
Issue acknowledged. The team resolved the finding by copping `epochData.ValidDealers` and recomputes `AggregatedShares` from consensus dealers. Changes have been reflected in the commit `4badffbef8e147b79396baf2847a4c555acc89cf`.

GEB-36 | Authority Mismatch In `MigrateAllWrappedTokenContracts()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	inference-chain/x/inference/keeper/bridge_wrapped_token.go (82c43a4): 427~450, 494~503	Resolved

Description

`MigrateAllWrappedTokenContracts()` will fail due to authorization mismatch, because wrapped token contracts are instantiated with the `governanceAddr` role (in `GetOrCreateWrappedTokenContract()`), but the migration uses `adminAddr`, which is derived from `k.AccountKeeper.GetModuleAddress(types.ModuleName)`, causing Denial of Service.

- **inference-chain/x/inference/keeper/bridge_wrapped_token.go**:

```
 494  @>  governanceAddr := k.GetAuthority()
// Governance module address for WASM admin
 495      instantiateMsg := BridgeTokenInstantiateMsg{
 496          ChainId:         chainId,
 497          ContractAddress: contractAddress,
 498          InitialBalances: []Balance{},
 499          Mint: &MintInfo{
 500              Minter: k.AccountKeeper.GetModuleAddress(types.ModuleName).String()
, // Inference module as minter
 501          },
 502  @>      Admin: &governanceAddr,
// Pass admin explicitly to avoid querying during instantiation
 503      }
```

```
427 @>  adminAddr := k.AccountKeeper.GetModuleAddress(types.ModuleName)
428     if len(migrateMsg) == 0 {
429         migrateMsg = json.RawMessage([]byte("{}"))
430     }
431
432     iter, err := k.WrappedTokenContractsMap.Iterate(ctx, nil)
433     if err != nil {
434         return err
435     }
436     defer iter.Close()
437
438     contracts, err := iter.Values()
439     if err != nil {
440         return err
441     }
442
443     var firstErr error
444     for _, contract := range contracts {
445         wrappedAddr := contract.WrappedContractAddress
446         // Execute migrate on the contract
447         _, err := permissionedKeeper.Migrate(
448             ctx,
449             sdk.MustAccAddressFromBech32(wrappedAddr),
450 @>          adminAddr,
451             newCodeID,
452             migrateMsg,
453         )
```

Recommendation

Recommend ensuring migration is called by the Governance module.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by updating wrapped-token contract migration to use the governance address as the migration admin, matching the admin used during wrapped-token instantiation. Changes have been reflected in the commit `8fbed2495ac6fe6b36aa234b92f24ca08243ebca` .

GEB-54 | Operational Risks In Smart Contracts

Category	Severity	Location	Status
Operational Risk, Centralization	**Medium**	**inference-chain/contracts/community-sale/src/contract.rs (82c43a4): 201~206; inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 97, 108~109; proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (82c43a4): 231, 266**	**Mitigated**

Description

The `bridge` / `WRAPPED GONKA ERC-20` contract exposes multiple privileged functions gated only by the `owner`. Compromise or misuse of that key enables arbitrary key rotation and contract status update, directly affecting cross-chain custody.

Privileged functions (BridgeContract.sol)

- `setGroupKey()`: `owner` role can set/overwrite group keys without transition signatures; controls which BLS keys are accepted.
- `resetToNormalOperation()`: `owner` role can reset the contract state to NORMAL_OPERATION.

Both CosmWasm contracts rely on single configured admins/creators to perform sensitive actions. Compromise or misuse of these keys enables unilateral changes to token metadata, sales parameters, pausing, and fund withdrawals, affecting users and bridge flows.

Privileged functions

`wrapped-token` (`contracts/wrapped-token/src/contract.rs`)

- `update_metadata()`: `creator` and `admin` role (governance module) is allowed to change name, symbol, decimals overrides.
- `UpdateMarketing()`, `UploadLogo()`: `admin` role is allowed to update CW20 marketing/logo routed to cw20-base.
- Mint/burn control follows cw20-base `minter` set at instantiate (currently the inference module onchain); whoever is minter can mint arbitrarily.
- `migrate()`: `admin` role (governance module) is allowed to migrate to new contract.

`instantiate` sets ADMIN (`governance`) and CREATOR (`inference module`); these roles persist.

`community-sale` (`contracts/community-sale/src/contract.rs`)

- `pause_contract()` / `resume_contract()`: `admin` role is allowed to halt or resume purchases.
- `update_buyer()`: `admin` role is allowed tochange designated buyer address.

- `update_price()`: `admin` role is allowed to set sale price.
- `withdraw_native_tokens()`: `admin` role is allowed to arbitrary native withdrawals to any recipient.
- `emergency_withdraw()`: `admin` role is allowed to drain full native balance to arbitrary recipient.
- `migrate()`: `admin` role is allowed to migrate to new contract.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR

- Remove the risky functionality.

Alleviation

[Gonka, 03/31/2026]:
We'd like to clarify the scope of the ETH-side owner privileges: both `setGroupKey()` and `resetToNormalOperation()` are only callable when the contract is in `ADMIN_CONTROL` state, which occurs exclusively in two edge cases: a prolonged key absence (30-day timeout) or a BLS key conflict requiring rapid resolution.

Once governance approves a key, the contract returns to `NORMAL_OPERATION` and these functions become inaccessible. On the Cosmos/Gonka side, the `admin` role is held by the `governance` module, meaning sensitive actions require DAO majority approval rather than a single key.

We will deploy the ETH bridge contract with a multisig wallet as owner to eliminate the single point of failure for admin-state scenarios. This has been documented in the contract README:

- https://github.com/gonka-ai/gonka/commit/fe423b36b2b512a48868a9fb3c91d0fd47ecc185

[CertiK, 03/31/2026]:
It is recommended to consider implementing the aforementioned measures to help mitigate the risk of centralized failure. Additionally, CertiK encourages the project team to periodically review and strengthen the private key security management practices for all addresses associated with centralized roles.

We will update the finding once the relevant on-chain transactions and multisig addresses have been deployed and verified.

[Gonka, 06/16/2026]:
The ownership of the bridge smart contract `0x972a7a92d92796a98801a8818bcf91f1648f2f68` on Ethereum was transferred to the multisig wallet `0x1d806d0e3c7df74c14a49d58de98c189614e1768` through transaction `0xccb42a7c25a4590291370baed193f0f109c4a2da0ced288a4b3ae5930d5df588`.

At the time of writing, the multisig wallet `0x1d806d0e3c7df74c14a49d58de98c189614e1768` is configured with a 2-of-3 signature threshold. According to its Safe configuration (`https://app.safe.global/settings/setup?safe=eth:0x1d806D0E3C7Df74C14A49d58De98c189614E1768`), the wallet is controlled by the following three owners:

- 0x3857b3D0aDDc3692979327662D22F0dDEdaCa5aD
- 0x82981d2b3132869c779b1154173BE526c84e4315
- 0x09f9D7fdDebe8C84276e4aE1b82847a5548073B2

As a result, any privileged operation of bridge smart contract `0x972a7a92d92796a98801a8818bcf91f1648f2f68` on Ethereum requiring multisig authorization must be approved by at least two of the three designated owners.

[CertiK, 06/16/2026]:
CertiK strongly encourages the project team to periodically review and strengthen the private key security management practices for all addresses associated with privileged roles.

GEB-55 BLS Genesis Export/Import Drops In-Flight DKG And Signing State

Category	Severity	Location	Status
Volatile Code, Inconsistency	Medium	inference-chain/x/bls/module/genesis.go (82c43a4): 11, 22; inference-chain/x/bls/types/genesis.go (82c43a4): 9	Resolved

Description

In `genesis.go`, `ExportGenesis()` / `InitGenesis()` persist only `params` and `ActiveEpochId`, and `types.GenesisState` (genesis.pb.go) contains no fields for "EpochBLSData", "threshold signing requests", "validation state of group public key" or the "expiration index" (keys.go prefixes). On export/import or restart, any active DKG epochs and queued threshold signing requests are lost, breaking continuity.

`inference-chain/x/bls/module/genesis.go`

```
11 func InitGenesis(ctx sdk.Context, k keeper.Keeper, genState types.GenesisState)
{
12     // this line is used by starport scaffolding # genesis/module/init
13     if err := k.SetParams(ctx, genState.Params); err != nil {
14         panic(err)
15     }
16
17     // Set the active epoch ID from genesis
18     k.SetActiveEpochID(ctx, genState.ActiveEpochId)
19 }
20
21 // ExportGenesis returns the module's exported genesis.
22 func ExportGenesis(ctx sdk.Context, k keeper.Keeper) *types.GenesisState {
23     genesis := types.DefaultGenesis()
24     genesis.Params = k.GetParams(ctx)
25
26     // Export the current active epoch ID
27     activeEpochID, found := k.GetActiveEpochID(ctx)
28     if found {
29         genesis.ActiveEpochId = activeEpochID
30     }
31
32     // this line is used by starport scaffolding # genesis/module/export
33
34     return genesis
35 }
```

Recommendation

Extend GenesisState to include all stored BLS state (epochs with dealer/verification data, group keys, signing requests, expiration index) and update ExportGenesis/InitGenesis to marshal/unmarshal them so state survives export/import.

Alleviation

[Gonka, 03/31/2026]:
Issue acknowledged. The team resolved the finding by adding the missing BLS states. Changes have been reflected in the commit `fc285ad1b12a0224470ae2fc32d46c4b40c3cb19` .

GEB-06 | Known Security Issue In Upstream Dependencies

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/go.mod (82c43a4): 32	Resolved

Description

Upstream Cometbft v0.38.17 is vulnerable to the following publicly disclosed security issue.

- https://github.com/cometbft/cometbft/security/advisories/GHSA-c32p-wcqj-j677
- https://github.com/cometbft/cometbft/security/advisories/GHSA-hrhf-2vcr-ghch

`inference-chain/go.mod`

```
32      github.com/cometbft/cometbft v0.38.17
```

Recommendation

Recommend upgrading Cometbft to latest stable version, v0.38.21.

Alleviation

[Gonka, 03/31/2026]:

Issue acknowledged. The team resolved the finding by upgrading Cometbft to v0.38.21. Changes have been reflected in the commit `df4f53a42086ee709a378fbaee508adf082ebcce` .

GEB-07 Weak Address Validation In `withdraw()` In `wrapped-token` Contract

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 180	Resolved

Description

`withdraw()` only checks that `destination_address` is non-empty, it does not validate Ethereum address format/length. Invalid destinations can permanently burn user funds.

`inference-chain/contracts/wrapped-token/src/contract.rs`

```
165 fn withdraw(
166     deps: DepsMut,
167     env: Env,
168     info: MessageInfo,
169     amount: Uint128,
170     destination_address: String,
171 ) -> Result<Response, ContractError> {
172     if amount.is_zero() {
173         return Err(ContractError::InsufficientFunds {
174             balance: 0,
175             required: 1,
176         });
177     }
178
179     // Validate destination address is not empty
180 @>  if destination_address.trim().is_empty() {
181         return Err(ContractError::Std(StdError::generic_err(
"destination_address cannot be empty")));
182     }
183
184     // Delegate to cw20-base burn
185     let mut resp = cw20_base_contract::execute(
186         deps,
187         env.clone(),
188         info.clone(),
189         cw20_base_msg::ExecuteMsg::Burn { amount },
190     ).map_err(|e| ContractError::Std(StdError::generic_err(e.to_string())))?;
191
192     // Create the bridge withdrawal message
193     let bridge_msg = create_bridge_withdrawal_msg(
194         env.contract.address.to_string(),
// creator (this contract - will be the transaction signer)
195         info.sender.to_string(),          // user_address (the caller)
196         amount.to_string(),               // amount
197         destination_address.clone(),      // destination_address
198     )?;
199
200     resp = resp
201         .add_message(bridge_msg)
202         .add_attribute("method", "withdraw")
203         .add_attribute("burn_amount", amount)
204         .add_attribute("destination_address", destination_address);
205
206     Ok(resp)
207 }
```

Recommendation

Recommend strengthening the validation of `destination_address`.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by strengthening the validation of `destination_address`. Changes have been reflected in the commit `8fbed2495ac6fe6b36aa234b92f24ca08243ebca`.

GEB-08 | `ADMIN` Role Cannot Not Be Updated

Category	Severity	Location	Status
Inconsistency	Minor	inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 58	Resolved

Description

The contract persists `ADMIN` once during instantiate, including a fallback to `info.sender` when `query_wasm_contract_info()` fails. That stored admin is never re-synced with the chain's actual wasm admin. If the chain admin differs at instantiation or later rotates, `UpdateMetadata()` authorization is evaluated against a stale address, incorrectly granting or denying control.

`inference-chain/contracts/wrapped-token/src/contract.rs`

```
 30 pub fn instantiate(
 31     deps: DepsMut,
 32     env: Env,
 33     info: MessageInfo,
 34     msg: InstantiateMsg,
 35 ) -> Result<Response, ContractError> {
 36
// Note: We don't set_contract_version here because cw20_base_contract::instantiate
 37
// will set it to "crates.io:cw20-base". Our migrate function handles this by
 38     // allowing migration from both "wrapped-token" and "crates.io:cw20-base".
 39
 40     // Save creator (instantiator = inference module) - controls operations
 41     CREATOR.save(deps.storage, &info.sender)?;
 42
 43
// Save admin (WASM admin = governance module) - controls marketing and metadata
 44
// Use admin from message if provided, otherwise try to query contract info,
 45
// falling back to sender if query fails (contract not registered yet during
instantiation)

 46     let admin_addr = if let Some(admin_str) = &msg.admin {
 47         deps.api.addr_validate(admin_str)?
 48     } else {
 49         match deps.querier.query_wasm_contract_info(&env.contract.address) {
 50             Ok(contract_info) => contract_info.admin.unwrap_or(info.sender.
clone()),
 51             Err(_) => {
 52                 // During instantiation, the contract may not be queryable yet
 53
// Fall back to sender - the actual admin will be set by the chain
 54                 info.sender.clone()
 55             }
 56         }
 57     };
 58 @>  ADMIN.save(deps.storage, &admin_addr)?;
 59 //...
```

Recommendation

Avoid caching admin or add a sync path, either (1) query wasm admin at each `UpdateMetadata()` call and compare against `info.sender`, or (2) add an `UpdateAdmin` execute message callable only by current chain admin to update stored `ADMIN`.

Alleviation

[Gonka, 03/20/2026]:

Issue acknowledged. The team resolved the finding by querying the admin at each `UpdateMetadata()` call and and

comparing it against `info.sender`. Changes have been reflected in the commit `6bec144d2a0d1c275fc5f5991d7c648b6e8240bd`.

GEB-09 Migration From Cw20-Base Leaves Required Wrapped-Token State Uninitialized

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 291	Resolved

Description

The wrapped-token `migrate()` function allows upgrades from both "wrapped-token" and "crates.io:cw20-base", but it only updates the contract version and never initializes wrapped-token-specific state (`ADMIN`, `CREATOR`, `BRIDGE_INFO`). When migrating from a cw20-base contract, those keys do not exist, so post-migration calls like `UpdateMetadata()` (loads ADMIN/CREATOR) and `BridgeInfo` queries (loads `BRIDGE_INFO`) fail, leaving the contract partially unusable even though the migration succeeds.

`inference-chain/contracts/wrapped-token/src/contract.rs`

```
282 pub fn migrate(
283     deps: DepsMut,
284     _env: Env,
285     _msg: Binary,
286 ) -> Result<Response, ContractError> {
287     let old = get_contract_version(deps.storage)
288         .map_err(|e| ContractError::Std(StdError::generic_err(e.to_string())))?
;
289
290     // Allow migration from both cw20-base (legacy) and wrapped-token contracts
291     if old.contract != CONTRACT_NAME && old.contract != "crates.io:cw20-base" {
292         return Err(ContractError::Std(StdError::generic_err(format!(
293             "wrong contract: expected {} or crates.io:cw20-base, got {}",
294             CONTRACT_NAME, old.contract
295         ))));
296     }
297
298     set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)
299         .map_err(|e| ContractError::Std(StdError::generic_err(e.to_string())))?
;
300
301     Ok(Response::new()
302         .add_attribute("action", "migrate")
303         .add_attribute("from_contract", old.contract)
304         .add_attribute("from_version", old.version)
305         .add_attribute("to_version", CONTRACT_VERSION))
306 }
```

Recommendation

In migrate, detect the old contract name and initialize missing wrapped-token state (derive CREATOR/ADMIN from message or chain admin, and set BRIDGE_INFO from migration params). Alternatively, restrict migration to only prior wrapped-token versions and reject cw20-base origins unless the migration message provides all required state.

Alleviation

[Gonka, 03/20/2026]:
Issue acknowledged. The team resolved the finding by disallowing cw20 migration. Changes have been reflected in the commit `6bec144d2a0d1c275fc5f5991d7c648b6e8240bd` .

GEB-10 Migration Of `community-sale` Lacks Compatibility Checks And State Validation

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/contracts/community-sale/src/contract.rs (82c43a4): 437	Resolved

Description

The community-sale `migrate()` entry point only reads the previous contract version and writes a new version. It does not validate the previous contract name/version nor confirm that required state (CONFIG) exists. A mistaken migration from an unrelated contract or a corrupted state will still "succeed," but subsequent calls that load CONFIG will error, leaving the contract unusable.

`inference-chain/contracts/community-sale/src/contract.rs`

```
437  pub fn migrate(deps: DepsMut, _env: Env, _msg: Binary) -> Result<Response,
ContractError> {
438      let old = get_contract_version(deps.storage)
439          .map_err(|e| ContractError::Std(StdError::msg(e.to_string())))?;
440      set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)
441          .map_err(|e| ContractError::Std(StdError::msg(e.to_string())))?;
442      Ok(Response::new()
443          .add_attribute("action", "migrate")
444          .add_attribute("from_version", old.version)
445          .add_attribute("to_version", CONTRACT_VERSION))
446  }
```

Recommendation

In migrate, verify the old contract name/version matches expected values and confirm CONFIG exists (e.g., CONFIG.may_load with an error if None). If migrating from a legacy version, include state initialization or a migration message that supplies required fields.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by adding the missing validations. Changes have been reflected in the commit `8fbed2495ac6fe6b36aa234b92f24ca08243ebca` .

GEB-18 Slot Donation Picks Under-Allocated Donor, Enabling Sybil Weight Inflation

Category	Severity	Location	Status
Logical Issue	Minor	inference-chain/x/bls/keeper/dkg_initiation.go (82c43a4): 300	Resolved

Description

In `findDonorIndex()`, when donors tie on assigned slots, the tie-break selects the smaller remainder (`ri.LT(rd)`), which typically corresponds to a participant already rounded down relative to its ideal weight. This confiscates a slot from the under-allocated party while preserving the over-allocated one. An attacker can create a zero-slot “dust” participant to trigger donation and bias ties so a victim with a small remainder donates to the attacker coalition, inflating attacker slots above their proportional weight.

inference-chain/x/bls/keeper/dkg_initiation.go

```
300 func findDonorIndex(assigned []int64, remainders []math.LegacyDec,
participants []types.ParticipantWithWeightAndKey) int {
301     donor := -1
302     for i, p := range participants {
303         if p.PercentageWeight.IsZero() {
304             continue
305         }
306         if assigned[i] <= 1 {
307             continue
308         }
309         if donor == -1 {
310             donor = i
311             continue
312         }
313
314         if assigned[i] > assigned[donor] {
315             donor = i
316             continue
317         }
318         if assigned[i] == assigned[donor] {
319             ri := remainders[i]
320             rd := remainders[donor]
321             if !ri.Equal(rd) {
322 @>                 if ri.LT(rd) {
323                       donor = i
324                 }
325                 continue
326             }
327             if participants[i].Address < participants[donor].Address {
328                 donor = i
329             }
330         }
331     }
332     return donor
333 }
```

For example, with `totalSlots = 12` and weights summing to 12 (V=5.01, A=4.99, S1=0.60, S2=0.40, H=1.00), floors assign V=5(r=0.01), A=4(r=0.99), S1=0(r=0.60), S2=0(r=0.40), H=1(r=0.00) leaving 2 slots that go to A and S1, producing V=5 and A=5; step(4) then must give S2 a slot and, because both donors tie at 5 slots, `findDonorIndex` executes `if !ri.Equal(rd) { if ri.LT(rd) { donor = i } }` and picks V (0.01 < 0.99) as donor, dropping V to 4 and raising S2 to 1 so the attacker controls A(5)+S1(1)+S2(1)=7 slots versus an ideal of 5.99. A smaller variant with `TotalSlots = 4` similarly yields Victim ideal 2.01 vs Attacker ideal 1.99, remainder distribution ties them at 2 slots, and a dust node triggers donation that wrongly takes from the Victim (remainder 0.01) rather than the Attacker (0.99), leaving the Victim with 1 slot while the Attacker retains 2 despite lower total weight.

Recommendation

Reverse the remainder tie-break so the donor is the participant with the larger remainder (more over-allocated) when assigned ties.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by choosing the donor with larger remainder. Changes have been reflected in the commit `f37f67cc4d102ff966e367f8d93d2f9bae2ccc3a` .

GEB-19 | Secret Shares Logged In `logging.Debug`

Category	Severity	Location	Status
Volatile Code	Minor	decentralized-api/internal/bls/verifier.go (82c43a4): 296	Resolved

Description

`performVerificationAndReconstruction()` logs the reconstructed slot share (`aggregatedShare.String()`), which is the secret key share for that slot. Even though this runs off-chain, emitting private key material to logs leaks the share to anyone with log access and undermines threshold security.

decentralized-api/internal/bls/verifier.go

```
293      logging.Debug(verifierLogTag+"Completed slot share reconstruction",
inferenceTypes.BLS,
294          "slotIndex", slotIndex,
295          "slotOffset", slotOffset,
296  @>      "finalShare", aggregatedShare.String())
```

Recommendation

Remove the secret from logs entirely (or redact), and if debugging is needed, log only non-sensitive metadata (slot index/offset) or a short hash under a guarded debug flag that cannot leak usable key material.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by removing the logging of `aggregatedShare` . Changes have been reflected in the commit `3b3d3baee99623d7930e750b0a5f2ff2890211d4` .

GEB-20 | Decoding Returns Zero If Fails

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/x/inference/keeper/bridge_utils.go (82c43a4): 26	Resolved

Description

The function `ethereumAddressToBytes()` silently truncates inputs >40 hex chars, drops a trailing nibble, left-pads with zeros for short inputs, and returns zeroed bytes on decode errors.

Malformed addresses are therefore accepted and mapped to unintended 20-byte values, making it impossible to detect invalid caller input and potentially misrouting funds or signatures.

inference-chain/x/inference/keeper/bridge_utils.go, `ethereumAddressToBytes()`

```
 26 func ethereumAddressToBytes(address string) []byte {
 27     // Remove 0x prefix if present
 28     addr := address
 29     if len(addr) >= 2 && addr[:2] == "0x" {
 30         addr = addr[2:]
 31     }
 32
 33     // Convert hex string to 20 bytes using encoding/hex
 34
// Truncate to at most 40 hex chars and ensure even length (ignore dangling nibble)
 35     maxLen := 40
 36     n := len(addr)
 37     if n > maxLen {
 38         n = maxLen
 39     }
 40     if n%2 == 1 {
 41         n--
 42     }
 43
 44     addrBytes := make([]byte, 20)
 45     if n <= 0 {
 46         return addrBytes
 47     }
 48     decoded := make([]byte, n/2)
 49     if _, err := hex.Decode(decoded, []byte(addr[:n])); err != nil {
 50         return addrBytes
 51     }
 52     copy(addrBytes, decoded)
 53     return addrBytes
 54 }
```

Recommendation

Enforce strict validation: strip 0x, require exactly 40 hex characters, decode with `hex.DecodeString`, and return an error on any length/hex failure.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by changing bridge address, chain ID, and amount encoding helpers to fail explicitly on malformed input instead of silently returning zeroed/truncated values, and by propagating those errors through both mint and withdrawal signing flows. Changes have been reflected in the commit `e1900a96993f69dd22d9de6a9f74cb44f938e2c7`.

GEB-21 Ineffective Polynomial Degree Check In `evaluatePolynomial()`

Category	Severity	Location	Status
Volatile Code	Minor	decentralized-api/internal/bls/dealer.go (82c43a4): 315~317	Resolved

Description

`evaluatePolynomial()` misses the case that the polynomial is of degree 1. Though it's unlikely in the current setup, it would lead to out-of-range panic.

`decentralized-api/internal/bls/dealer.go`

```
314 func evaluatePolynomial(polynomial []*fr.Element, x uint32) *fr.Element {
315 @>    if len(polynomial) == 0 {
316         return new(fr.Element).SetZero()
317     }
318
319     // Convert x to fr.Element
320     xFr := new(fr.Element).SetUint64(uint64(x))
321
322     // Start with highest degree coefficient
323     result := new(fr.Element).Set(polynomial[len(polynomial)-1])
324
325     // Apply Horner's method: result = result * x + coeff[i]
326 @>    for i := len(polynomial) - 2; i >= 0; i-- {
327         result.Mul(result, xFr)
328         result.Add(result, polynomial[i])
329     }
330
331     return result
332 }
```

Recommendation

Recommend handling the degree 1 case.

Alleviation

[Gonka, 03/20/2026]:

Issue acknowledged. The team resolved the finding by handling the polynomial of degree 1. Changes have been reflected in the commit `ca87b54e516c4d40f61bf6620d1c8d8193cca2b5` .

GEB-22 | Unchecked `amountToBytes32()` Panic On Oversized Amounts

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/x/inference/keeper/bridge_utils.go (82c43a4): 57, 67	Resolved

Description

Both `amountToBytes32()` and `chainIdToBytes32()` call `FillBytes()` on a fixed 32-byte buffer without checking the numeric size of the input. If a user supplies an amount that exceeds 256 bits, `FillBytes()` will panic.

This helper is invoked in bridge message handlers in request mint/withdraw paths, so an oversized amount in a transaction can trigger a panic during message processing.

While Cosmos SDK typically recovers panics and aborts the tx, this creates a crash path, degrades observability, and risks node instability if panic recovery is bypassed elsewhere.

`inference-chain/x/inference/keeper/bridge_utils.go`

```
57 func chainIdToBytes32(chainId string) []byte {
58     chainIdBytes := make([]byte, 32)
59     if chainIdInt, ok := math.NewIntFromString(chainId); ok {
60         chainIdBigInt := chainIdInt.BigInt()
61 @>          chainIdBigInt.FillBytes(chainIdBytes) // Big endian format
62     }
63     return chainIdBytes
64 }
65
66 // amountToBytes32 converts an amount string to bytes32 format (uint256)
67 func amountToBytes32(amount string) []byte {
68     amountBytes := make([]byte, 32)
69     if amountInt, ok := math.NewIntFromString(amount); ok {
70         amountBigInt := amountInt.BigInt()
71 @>          amountBigInt.FillBytes(amountBytes) // Big endian format
72     }
73     return amountBytes
74 }
```

Recommendation

Enforce bounds before calling `FillBytes()`: parse the amount, reject values whose byte length exceeds 32 (or greater than 2^256-1), and return an error from the handler.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding explicit negative-value and 256-bit size checks before calling

`FillBytes()` in both `chainIdToBytes32()` and `amountToBytes32()`, so oversized inputs now fail cleanly instead of triggering a panic. Changes have been reflected in the commit `f0ecf522ceab1862234b88379fdf7b12bbc3f7b6`.

GEB-23 Missing Validation Of `MsgRequestThresholdSignature.ValidateBasic()` For `chain_id` / `request_id` And Data Chunk Sizes

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/x/bls/types/message_request_threshold_signature.go (82c43a4): 11	Resolved

Description

`MsgRequestThresholdSignature.ValidateBasic()` only checks creator, epoch, and non-empty data, but the proto contracts declare `ChainId` and `RequestId` as 32-byte fields and data as 32-byte chunks. Currently a user can submit zero-length or wrong-sized `ChainId` / `RequestId`, or arbitrarily sized data chunks. These malformed inputs flow to `encodeSigningData` (which just concatenates) and to storage keys, producing non-EVM-compatible payloads and inconsistent signing hashes. At best the request will be unusable; at worst it can bloat state or create ambiguous identifiers.

`inference-chain/x/bls/types/message_request_threshold_signature.go`

```
 11 func (m *MsgRequestThresholdSignature) ValidateBasic() error {
 12     if _, err := sdk.AccAddressFromBech32(m.Creator); err != nil {
 13         return errorsmod.Wrap(sdkerrors.ErrInvalidAddress,
"invalid creator address")
 14     }
 15     if m.CurrentEpochId == 0 {
 16         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"current_epoch_id must be > 0")
 17     }
 18     if len(m.Data) == 0 {
 19         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"data must be non-empty")
 20     }
 21     return nil
 22 }
```

Recommendation

Strengthen `ValidateBasic()` to enforce the expected shapes:

- len(`ChainId`) == 32
- len(`RequestId`) == 32
- len(`Data`) > 0 and every element len(`Data[i]`) == 32
- bound the number of data elements to a sane limit to avoid large payloads.

Return error on any violation so malformed requests are rejected before hitting the keeper.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by adding the missing validations. Changes have been reflected in the commit `dc54bf10fd4d9e8a99329d2c12fe45f59cec26f1` .

GEB-24 Insufficient Validation For Dealer Part Submissions In `MsgSubmitDealerPart.ValidateBasic()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/x/bls/types/message_submit_dealer_part.go (82c43a4): 11	Resolved

Description

`MsgSubmitDealerPart.ValidateBasic()` only checks creator, epoch non-zero, and that commitments and `EncryptedSharesForParticipants` are non-empty. It does not enforce expected sizes for commitments (compressed G2 points), non-zero values, per-participant share presence/size, or any upper bounds on counts. Malformed or oversized payloads can enter the DKG pipeline, leading to failed deserialization/verification, potential state bloat, and denial of service during dealing.

`inference-chain/x/bls/types/message_submit_dealer_part.go`

```
 11 func (m *MsgSubmitDealerPart) ValidateBasic() error {
 12     // creator address
 13     if _, err := sdk.AccAddressFromBech32(m.Creator); err != nil {
 14         return errorsmod.Wrap(sdkerrors.ErrInvalidAddress,
"invalid creator address")
 15     }
 16     // epoch id
 17     if m.EpochId == 0 {
 18         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"epoch_id must be > 0")
 19     }
 20     // commitments: non-empty, each G2 size and non-zero bytes
 21     if len(m.Commitments) == 0 {
 22         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"commitments must be non-empty")
 23     }
 24
// encrypted shares for participants: non-empty, bounded, and each entry non-empty
with non-empty shares

 25     if len(m.EncryptedSharesForParticipants) == 0 {
 26         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"encrypted_shares_for_participants must be non-empty")
 27     }
 28     return nil
 29 }
```

Recommendation

Harden `ValidateBasic()` with strict structural checks and bounds:

- Enforce expected commitments count and each commitment to be the expected compressed G2 length and non-zero.
- Require `EncryptedSharesForParticipants` count within limits; each participant entry must be non-empty, with each share the expected size (field element length).
- Optionally cap payload sizes to prevent excessive gas/processing.

Reject any violations with error to drop malformed dealer parts before they reach keeper logic.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by adding the missing validations. Changes have been reflected in the commit `3a99afbc15a51b8d175368c31cbc5a47a1317f68` .

GEB-25 Missing Validation In Group Key Validation Signatures In `MsgSubmitGroupKeyValidationSignature.ValidateBasic()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/x/bls/types/message_submit_group_key_validation_signature.go (82c43a4): 11	Resolved

Description

`MsgSubmitGroupKeyValidationSignature.ValidateBasic()` only checks creator format, `NewEpochId` > 0, and that `SlotIndices` is non-empty. It does not validate the signature fields themselves (presence and expected lengths) nor bounds on `SlotIndices`. Malformed or overlong payloads can pass basic checks, leading to downstream errors or excessive processing.

`inference-chain/x/bls/types/message_submit_group_key_validation_signature.go`

```
 11 func (m *MsgSubmitGroupKeyValidationSignature) ValidateBasic() error {
 12     if _, err := sdk.AccAddressFromBech32(m.Creator); err != nil {
 13         return errorsmod.Wrap(sdkerrors.ErrInvalidAddress,
"invalid creator address")
 14     }
 15     if m.NewEpochId == 0 {
 16         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"new_epoch_id must be > 0")
 17     }
 18     if len(m.SlotIndices) == 0 {
 19         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"slot_indices must be non-empty")
 20     }
 21     return nil
 22 }
```

Recommendation

Extend `ValidateBasic()` to reject malformed messages:

- Ensure len(`PartialSignature`) match expected sizes and are non-zero.
- Bound `SlotIndices` count and ensure elements are non-zero and within expected range.

Return error on any violation so invalid submissions are dropped early.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the missing validations. Changes have been reflected in the commit `c2fcc012a4da6918c6ac1285cda621447acc9d9b` .

GEB-26 Missing Validation In Partial Signature Submissions In `MsgSubmitPartialSignature.ValidateBasic()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/x/bls/types/message_submit_partial_signature.go (82c43a4): 11	Resolved

Description

`MsgSubmitPartialSignature.ValidateBasic()` only checks the creator address and that `SlotIndices` is non-empty. It does not validate the partial signature bytes (presence/expected length), nor bound/validate slot_indices contents. Malformed or oversized payloads can pass basic checks, leading to downstream errors during aggregation and potential denial of service from large index lists.

`inference-chain/x/bls/types/message_submit_partial_signature.go`

```
11  func (m *MsgSubmitPartialSignature) ValidateBasic() error {
12      if _, err := sdk.AccAddressFromBech32(m.Creator); err != nil {
13          return errorsmod.Wrap(sdkerrors.ErrInvalidAddress,
"invalid creator address")
14      }
15      if len(m.SlotIndices) == 0 {
16          return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"slot_indices must be non-empty")
17      }
18      return nil
19  }
```

Recommendation

Strengthen `ValidateBasic()` to enforce:

- len(`PartialSignature`) equals the expected BLS compressed size and is non-zero.
- `SlotIndices` count within protocol limits; each index non-zero and within the valid participant slot range.
- Optionally cap overall message size.

Reject violations with error so invalid submissions are dropped before hitting keeper logic.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the missing validations. Changes have been reflected in the commit `d89272c175cf2b6716242992c640babfa298089a` .

GEB-27 Unbounded `DealerValidity` In Verification Vector Submissions Of `MsgSubmitVerificationVector.ValidateBasic()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/x/bls/types/message_submit_verification_vector.go (82c43a4): 11	Resolved

Description

`MsgSubmitVerificationVector.ValidateBasic()` only checks that `DealerValidity` is non-empty and does not cap or align its length to the expected participant count. A malicious sender can submit an arbitrarily long bitmap, inflating message size and wasting processing, or a mismatched length that doesn't correspond to the epoch's dealers, leading to ambiguous or ignored validity data.

`inference-chain/x/bls/types/message_submit_verification_vector.go`

```
 11  func (m *MsgSubmitVerificationVector) ValidateBasic() error {
 12      if _, err := sdk.AccAddressFromBech32(m.Creator); err != nil {
 13          return errorsmod.Wrap(sdkerrors.ErrInvalidAddress,
"invalid creator address")
 14      }
 15      if m.EpochId == 0 {
 16          return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"epoch_id must be > 0")
 17      }
 18      if len(m.DealerValidity) == 0 {
 19          return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"dealer_validity must be non-empty")
 20      }
 21      return nil
 22  }
```

Recommendation

Bound and align `DealerValidity` in `ValidateBasic()`:

- Require len(DealerValidity) > 0 and ≤ a protocol-defined maximum (ideally the known participant count for the epoch).
- Optionally enforce exact equality to the epoch's dealer list length in the handler.

Reject violations with error to prevent oversized or mismatched payloads from entering DKG processing.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the missing validations. Changes have been reflected in the commit `d89272c175cf2b6716242992c640babfa298089a` .

GEB-37 | DKG Process Can Be Stuck Due To Internal Errors

Category	Severity	Location	Status
Logical Issue, Design Issue	Minor	inference-chain/x/bls/keeper/phase_transitions.go (82c43a4): 41	Resolved

Description

`ProcessDKGPhaseTransitionForEpoch()` returns on the first error from `TransitionToVerifyingPhase` / `CompleteDKG`. In the `CompleteDKG` path, errors from `DetermineValidDealersWithConsensus` or `ComputeGroupPublicKey` propagate up, and the epoch state remains unchanged (still VERIFYING, active epoch not cleared, phase not set to FAILED). Because `EndBlock` just logs and ignores the error, the same epoch will be retried every block and stay stuck, blocking progression/liveness.

inference-chain/x/bls/keeper/phase_transitions.go

```
 41      switch epochBLSData.DkgPhase {
 42      case types.DKGPhase_DKG_PHASE_DEALING:
 43          if currentBlockHeight >= epochBLSData.DealingPhaseDeadlineBlock {
 44              if err := k.TransitionToVerifyingPhase(ctx, &epochBLSData); err !=
nil {
 45                  return fmt.Errorf(
"failed to transition DKG to verifying phase for epoch %d: %w", epochID, err)
 46              }
 47          }
 48      case types.DKGPhase_DKG_PHASE_VERIFYING:
 49          if currentBlockHeight >= epochBLSData.VerifyingPhaseDeadlineBlock {
 50              if err := k.CompleteDKG(ctx, &epochBLSData); err != nil {
 51                  return fmt.Errorf("failed to complete DKG for epoch %d: %w",
 epochID, err)
 52              }
 53          }
 54      }
```

```
171      validDealers, err := k.DetermineValidDealersWithConsensus(epochBLSData)
172      if err != nil {
173          return fmt.Errorf("failed to determine valid dealers for epoch %d: %w",
 epochBLSData.EpochId, err)
174      }
175
176      groupPublicKey, err := k.ComputeGroupPublicKey(epochBLSData, validDealers)
177      if err != nil {
178          return fmt.Errorf("failed to compute group public key for epoch %d: %w"
, epochBLSData.EpochId, err)
179      }
```

Recommendation

On unrecoverable errors inside `CompleteDKG` / `TransitionToVerifyingPhase`, explicitly mark the epoch `DKG_PHASE_FAILED`, store it, and clear the active epoch to allow future rounds to proceed; alternatively, catch errors in `ProcessDKGPhaseTransitionForEpoch()` and transition to FAILED with a reason.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by introducing a unified DKG failure handler and routing internal DKG completion errors through it, so failed epochs are now marked as `FAILED`, persisted, and removed from the active DKG slot instead of being retried indefinitely every block. Changes have been reflected in the commit `c991b7aff69b5d23f0c714e185ea3cfddeb9594e`.

GEB-38 | Inconsistent Comparison Of Deadline Block

Category	Severity	Location	Status
Coding Issue	Minor	inference-chain/x/bls/keeper/msg_server_dealer.go (82c43a4): 27~29; inference-chain/x/bls/keeper/msg_server_verifier.go (82c43a4): 31~33	Resolved

Description

The comparison between current block height and the deadline block height is inconsistent across the different modules.

inference-chain/x/bls/keeper/msg_server_dealer.go

```
27  @>  if ctx.BlockHeight() > epochBLSData.DealingPhaseDeadlineBlock {
28          return nil, fmt.Errorf("dealing phase deadline has passed for epoch %d",
 msg.EpochId)
29      }
```

inference-chain/x/bls/keeper/msg_server_verifier.go

```
30      currentHeight := sdkCtx.BlockHeight()
31  @>  if currentHeight >= epochBLSData.VerifyingPhaseDeadlineBlock {
32          return nil, status.Error(codes.DeadlineExceeded, fmt.Sprintf(
"verification deadline passed: current height %d >= deadline %d", currentHeight,
 epochBLSData.VerifyingPhaseDeadlineBlock))
33      }
```

Recommendation

Recommend ensuring all checks are consistent.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by aligning verification-vector deadline checks with the rest of the BLS submission flow, so submissions are now accepted through the deadline block and only rejected after it has passed.

Changes have been reflected in the commit `1038dbd818be33d9453815e2f7a1cd2546d64d6a` .

GEB-39 Missing Validation Of `msg.Amount` Being Positive In `MsgRequestBridgeWithdrawal`

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/x/inference/types/message_request_bridge_withdrawal.go (82c43a4): 34	Resolved

Description

`MsgRequestBridgeWithdrawal.ValidateBasic()` only check the `Amount` is not empty, instead of checking its value being positive.

`inference-chain/x/inference/types/message_request_bridge_mint.go`

```
 20 func (msg *MsgRequestBridgeWithdrawal) ValidateBasic() error {
 21     // Validate creator address (contract signer)
 22     _, err := sdk.AccAddressFromBech32(msg.Creator)
 23     if err != nil {
 24         return errorsmod.Wrapf(sdkerrors.ErrInvalidAddress,
"invalid creator address (%s)", err)
 25     }
 26
 27     // Validate user address
 28     _, err = sdk.AccAddressFromBech32(msg.UserAddress)
 29     if err != nil {
 30         return errorsmod.Wrapf(sdkerrors.ErrInvalidAddress,
"invalid user address (%s)", err)
 31     }
 32
 33     // Validate amount is not empty
 34     if len(msg.Amount) == 0 {
 35         return errorsmod.Wrap(sdkerrors.ErrInvalidRequest,
"amount cannot be empty")
 36     }
 37 //...
```

Recommendation

Recommend strengthening the validation of `Amount` to be positive.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the missing validation. Changes have been reflected in the

commit `8fbed2495ac6fe6b36aa234b92f24ca08243ebca` .

GEB-40 | Broken Cleanup Logic

Category	Severity	Location	Status
Logical Issue, Coding Issue	Minor	inference-chain/x/inference/keeper/bridge_transaction.go (82c43a4): 114	Resolved

Description

`CleanupOldBridgeTransactions()` is used to remove the bridge transactions older than the specified block number. However, it directly uses the Go language's string comparison operator `<` .

inference-chain/x/inference/keeper/bridge_transaction.go, `CleanupOldBridgeTransactions()`

```
  99  func (k Keeper) CleanupOldBridgeTransactions(ctx context.Context, chainId
string, maxBlockNumber string) (int, error) {
 100      iter, err := k.BridgeTransactionsMap.Iterate(ctx, collections.
NewPrefixedTripleRange[string, string, string](chainId))
 101      if err != nil {
 102          return 0, err
 103      }
 104      defer iter.Close()
 105
 106      values, err := iter.Values()
 107      if err != nil {
 108          return 0, err
 109      }
 110
 111      var deletedCount int
 112      var firstErr error
 113      for _, tx := range values {
 114  @>      if tx.BlockNumber < maxBlockNumber {
 115              if err := k.removeBridgeTransactionByID(ctx, tx.Id); err != nil {
 116      ...
```

String comparison follows lexicographical order, if `tx.BlockNumber = "1000"` and `maxBlockNumber = "99"` , then the if condition is true, breaking the cleanup logic(incorrectly cleanup the new bridge transaction).

In addition, the comments said "This is efficient because block numbers are included in the key prefix", but it did not use prefixes for range queries (e.g., `Range (Start, End)`), but instead performed a full table scan.

Note that `CleanupOldBridgeTransactions()` is not used in the current codebase.

Recommendation

Recommend transfering string to uint before comparison.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by replacing lexicographical block-number comparison with numeric parsing/comparison in `CleanupOldBridgeTransactions()`, and by correcting the implementation comment to reflect that the function currently performs a full scan rather than an efficient block-range query. Changes have been reflected in the commit `159253b69daab28fc6b6035adf8a4863a27551f2`.

GEB-42 | Unhandled Error Of `EmitTypedEvent()`

Category	Severity	Location	Status
Inconsistency	Minor	inference-chain/x/bls/keeper/dkg_initiation.go (82c43a4): 77; inference-chain/x/bls/keeper/msg_server_group_validation.go (82c43a4): 69; inference-chain/x/bls/keeper/msg_server_verifier.go (82c43a4): 72	Resolved

Description

In the linked places, the `EmitTypedEvent()` does not handle the return error, which would lead to the off-chain relayer misses the emitted events.

Recommendation

Recommend handling the error.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by handling `EmitTypedEvent()` return values in the affected BLS keeper flows, so event emission failures now propagate explicitly instead of being silently ignored. Changes have been reflected in the commit `1fb76ae2e95512f721a549f109718a2d3aee740981` .

[CertiK, 03/20/2026]:
In the commit `1fb76ae2e95512f721a549f109718a2d3aee740981` , the `SubmitDealerPart()` only logs the error instead of returning it:

```
if err := ctx.EventManager().EmitTypedEvent(event); err != nil {
    ms.Logger().Error("Failed to emit EventDealerPartSubmitted", "error", err)
}
```

[Gonka, 04/03/2026]:
The reason it was left non-fatal: preserve tx liveness for a non-consensus side effect (events), especially since state write already happened just above at msg_server_dealer.go:84. Returning error there would fail the tx (and rollback cached state), potentially blocking dealer submissions over an observability issue.

GEB-43 | Valid Dealers Can Be Less Than Threshold

Category	Severity	Location	Status
Logical Issue	Minor	inference-chain/x/bls/keeper/phase_transitions.go (82c43a4): 169~171	Resolved

Description

The function `CompleteDKG()` starts to computing valid dealers when `slotsWithVerification` exceeding `epochBLSData.ITotalSlots/2`, however, it did not double check whether the number of valid dealers exceeded the threshold.

inference-chain/x/bls/keeper/phase_transitions.go, `CompleteDKG()`

```
 168    if slotsWithVerification > epochBLSData.ITotalSlots/2 {
 169
// Sufficient verification participation - compute group public key using dealer
consensus

 170  @>    validDealers, err := k.DetermineValidDealersWithConsensus(epochBLSData)
 171        if err != nil {
 172            return fmt.Errorf(
"failed to determine valid dealers for epoch %d: %w", epochBLSData.EpochId, err)
 173        }
 174    ...
```

As an result, the final `validDealers` array may not meet the threshold, causing an entropy missing issue.

Recommendation

Recommend checking the `validDealers` again, ensuring there are enough valid dealers.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by adding a second threshold check after dealer consensus is computed, ensuring that DKG only proceeds when the resulting valid dealers cover more than the required slot threshold; otherwise the epoch is marked as failed instead of continuing to group key generation. Changes have been reflected in the commit `e149bb777d2c74db836ad4bbbe8f3fff7c50b2f5`.

GEB-47 | Hard-Coded Threshold For BLS Signature

Category	Severity	Location	Status
Logical Issue	Minor	inference-chain/x/bls/keeper/msg_server_group_validation.go (82c43a4): 176~177	Resolved

Description

In validating signature from previous epoch, the `requiredSlots` is hard-coded to be half of the `ITotalSlots`. Ideally, it should be constructed from `TSlotsDegree` to reflect future change of parameters.

`inference-chain/x/bls/keeper/msg_server_group_validation.go`

```
176      requiredSlots := previousEpochBLSData.ITotalSlots/2 + 1
```

Recommendation

Compute `requiredSlots` from `TSlotsDegree` in `previousEpochBLSData`.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by replacing the hard-coded previous-epoch signature threshold with the epoch-specific reconstruction threshold `TSlotsDegree + 1` in the group key validation flow. Changes have been reflected in the commit `1d20a51fb52e300def6efd60ce4bb46898274b98`.

GEB-48 | Missing Signed Status In `parseEpochDataFromJSON()`

Category	Severity	Location	Status
Volatile Code	Minor	decentralized-api/internal/bls/verifier.go (82c43a4): 654~664	Resolved

Description

`parseEpochDataFromJSON()` misses handling of the `signed` status:

`decentralized-api/internal/bls/verifier.go`, `parseEpochDataFromJSON()`

```
653     if dkgPhaseStr, ok := epochDataMap["dkg_phase"].(string); ok {
654         switch dkgPhaseStr {
655         case "DKG_PHASE_UNDEFINED":
656             epochDataMap["dkg_phase"] = int32(0)
657         case "DKG_PHASE_DEALING":
658             epochDataMap["dkg_phase"] = int32(1)
659         case "DKG_PHASE_VERIFYING":
660             epochDataMap["dkg_phase"] = int32(2)
661         case "DKG_PHASE_COMPLETED":
662             epochDataMap["dkg_phase"] = int32(3)
663         case "DKG_PHASE_FAILED":
664             epochDataMap["dkg_phase"] = int32(4)
665         default:
666             // Try to parse as number if it's a numeric string
667             if dkgPhaseNum, err := strconv.ParseUint(dkgPhaseStr, 10, 32); err
== nil {
668                 epochDataMap["dkg_phase"] = int32(dkgPhaseNum)
669             }
670         }
671     }
```

Recommendation

Recommend adding the `signed` case.

Alleviation

[Gonka, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the `signed` case. Changes have been reflected in the commit `67792bb1c50632d8e4ad663d8c7e1259749042b3`.

GEB-49 | Missing Check Of Withdrawal And Mint Amount

Category	Severity	Location	Status
Volatile Code	Minor	proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (82c43a4): 354, 383	Resolved

Description

In both `withdraw()` and `mintWithSignature()`, the input `amount` has not been validated to be positive.

Recommendation

Recommend adding the check to ensure they are positive.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by adding explicit non-zero amount checks to both `withdraw()` and `mintWithSignature()` in the Ethereum bridge contract, rejecting zero-value commands before execution. Changes have been reflected in the commit `1613d70be3d18d294f12f5297522aabb41f810ce`.

GEB-56 Missing Validation Of Epoch Id In `RequestThresholdSignature()`

Category	Severity	Location	Status
Volatile Code, Inconsistency	Minor	inference-chain/x/bls/keeper/msg_server_threshold_signing.go (82c43a4): 53	Resolved

Description

`RequestThresholdSignature()` handles requests for threshold signatures from external users, however, it does not check if the input `msg.CurrentEpochId` matches the current epoch Id, so external callers can request signatures against any stored completed epoch (including stale epochs after rotation).

`inference-chain/x/bls/keeper/msg_server_threshold_signing.go`

```
 46 func (ms msgServer) RequestThresholdSignature(ctx context.Context, msg *types.
MsgRequestThresholdSignature) (*types.MsgRequestThresholdSignatureResponse, error) {
 47     // Convert to SDK context
 48     sdkCtx := sdk.UnwrapSDKContext(ctx)
 49
 50
// Add domain separation for external requests by prepending CUSTOM_SIGNATURE_DOMAIN
 51
// This prevents unauthorized signatures from being created for unintended
operations

 52     customData := make([][]byte, 0, len(msg.Data)+1)
 53     customData = append(customData, CUSTOM_SIGNATURE_DOMAIN)
 54     customData = append(customData, msg.Data...)
 55
 56     // Create SigningData struct from the message with custom domain prefix
 57     signingData := types.SigningData{
 58 @>         CurrentEpochId: msg.CurrentEpochId,
 59         ChainId:        msg.ChainId,
 60         RequestId:      msg.RequestId,
 61         Data:           customData,
 62     }
 63
 64     // Call the core RequestThresholdSignature function which handles:
 65     // 1. Validates the request (epoch, uniqueness, etc.)
 66     // 2. Creates and stores the ThresholdSigningRequest
 67     // 3. Emits EventThresholdSigningRequested for controllers
 68     err := ms.Keeper.RequestThresholdSignature(sdkCtx, signingData)
 69     if err != nil {
 70         return nil, fmt.Errorf("failed to request threshold signature: %w", err
)
 71     }
 72
 73     return &types.MsgRequestThresholdSignatureResponse{}, nil
 74 }
```

Recommendation

Recommend validating the input `msg.CurrentEpochId` matches the current epoch Id.

Alleviation

[Gonka, 03/20/2026]:

Issue acknowledged. The team resolved the finding by adding the validation of epoch Id. Changes have been reflected in the commit `1ebd20022e6c03388cd8a4c7646453ca5bdef0da` .

GEB-57 | Slot Range Silently Clamped Instead Of Failing

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/x/bls/keeper/dkg_initiation.go (82c43a4): 268~270	Resolved

Description

In `AssignSlots()` of `dkg_initiation.go`, `endIndex` is clamped to `totalSlots-1` even though `checkTotal` already guarantees assigned counts sum to `totalSlots`. If this branch is hit, it signals a slot misallocation; clamping masks the bug and could leave overlapping or missing slots undetected.

`inference-chain/x/bls/keeper/dkg_initiation.go`, `AssignSlots()`

```
260     for i, participant := range sortedParticipants {
261         slotCount := assigned[i]
262         if slotCount <= 0 {
263             continue
264         }
265
266         startIndex := currentSlot
267         endIndex := startIndex + uint32(slotCount) - 1
268         if endIndex >= totalSlots {
269             endIndex = totalSlots - 1
270         }
271 //...
```

Recommendation

Replace the clamp with a guard that returns an error when `endIndex >= totalSlots` so invalid allocations fail fast.

Alleviation

[Gonka, 03/20/2026]:

Issue acknowledged. The team resolved the finding by returning error instead of clamping. Changes have been reflected in the commit `02f1db477ddc383f369772e18400a66f9ba7def7`.

GEB-59 `wrapped-token` Instances Are Not Migratable Because Deployment Leaves CW2 Marker As `crates.io:cw20-base`

Category	Severity	Location	Status
Inconsistency	Minor	inference-chain/contracts/wrapped-token/src/contract.rs (04/21-6d0dee1): 321	Resolved

Description

`instantiate()` delegates initialization to `cw20_base_contract::instantiate()` and does not overwrite the CW2 contract metadata afterward, so deployed instances retain `crates.io:cw20-base`. However, `migrate()` only accepts `wrapped-token`.

```
313 pub fn migrate(
314     deps: DepsMut,
315     _env: Env,
316     _msg: Binary,
317 ) -> Result<Response, ContractError> {
318     let old = get_contract_version(deps.storage)
319         .map_err(|e| ContractError::Std(StdError::generic_err(e.to_string())))?
;
320
321 @>  if old.contract != CONTRACT_NAME {
322         return Err(ContractError::Std(StdError::generic_err(format!(
323             "wrong contract: expected {}, got {}",
324             CONTRACT_NAME, old.contract
325         ))));
326     }
327 //...
```

As a result, wrapped-token contracts created through the normal flow fail their own upgrade check and revert during migration.

Recommendation

After the delegated instantiate call, write the correct CW2 metadata with `set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)`.

Alleviation

[Gonka, 05/13/2026]:

Issue acknowledged. Changes have been reflected in the commit `5ad7b4c0a9b7adb3e5cf1c81627b3cf14cb7baa2`.

GEB-60 | `WGNKBurned` Can Be Emitted During `ADMIN_CONTROL`

Category	Severity	Location	Status
Inconsistency	Minor	proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (04/21-6d0dee1): 288, 448, 466	Resolved

Description

The bridge pauses `withdraw()` and `mintWithSignature()` with `onlyNormalOperation`, but it does not apply the same check to the auto-burn paths in `transfer()` and `transferFrom()` when `to == address(this)`. So users and approved spenders can still burn WGNK and emit WGNKBurned while the contract is in `ADMIN_CONTROL`.

This bypasses the intended emergency-stop behavior for the reverse bridge. If off-chain relayers honor WGNKBurned during ADMIN_CONTROL, native asset releases from escrow may continue even though the bridge is supposed to be paused.

Additionally, `submitGroupKey()` is also not enforced with `onlyNormalOperation`.

Recommendation

Require `NORMAL_OPERATION` for all bridge-consumable burn paths.

Alleviation

[Gonka, 05/13/2026]:
Issue acknowledged. Changes have been reflected in the commit `75d552e12faf0f5d56e24522fcdedf26cedca4c7`.

GEB-61 Uncapped Warm-Key Fanout Can Make Dealer Parts Unsendable And Stall DKG

Category	Severity	Location	Status
Volatile Code	Minor	inference-chain/x/inference/module/module.go (04/21-6d0dee1): 1582	Resolved

Description

The BLS snapshot includes one primary secp256k1 key plus every authz-derived warm key in `AllowedSecp256K1PublicKeys`, with no independent cap in the collection path. Dealer generation then emits one encrypted share per slot per allowed key, while chain-side shape validation requires that exact snapshot-derived count. At the same time, `MsgSubmitDealerPart.ValidateBasic()` enforces a hard limit of 16384 encrypted shares for each participant row.

This creates a hard liveness failure. If an active participant accumulates enough warm keys that `slotCount * (1 + len(AllowedSecp256K1PublicKeys)) > 16384`, every dealer is forced to build an oversized ciphertext array for that participant, and the dealer part becomes invalid before execution. Because all dealers must include shares for every participant, a single participant with excessive warm-key fanout can block dealer-part submission and stall the DKG round.

Recommendation

Enforce a consistent upper bound on warm-key fanout before DKG begins. The safest fix is to cap `AllowedSecp256K1PublicKeys` so that, for every participant, `slotCount * (1 + len(AllowedSecp256K1PublicKeys)) <= 16384` always holds.

Also align all layers to the same limit:

- Reject or prune excess warm keys when building the BLS participant snapshot.
- Refuse to initiate DKG if any participant's derived encrypted-share count exceeds the message bound.
- Keep dealer generation and chain validation synchronized to the same invariant.

Alleviation

[Gonka, 05/13/2026]:
Issue acknowledged. Changes have been reflected in the commit `8d98de2c8fa06ef58b1f2ca421b4912786472be9`.

GEB-62 Candidate Dealer Selection Occurs Before Complaint Adjudication

Category	Severity	Location	Status
Inconsistency, Logical Issue	Minor	inference-chain/x/bls/keeper/phase_transitions.go (04/21-6d0dee1): 144, 368	Resolved

Description

During the VERIFYING-to-DISPUTING transition, dealer candidacy is determined solely from raw verifier votes in phase_transitions.go and stored as `CandidateValidDealers` in phase_transitions.go. Complaints against dealers that do not make this vote-based candidate set are then filtered out and deleted in phase_transitions.go, while dealers can only respond to complaints that still exist in msg_server_dispute.go. The substantive truth check for a complaint is deferred until `verifyDealerComplaintResponse` in dispute_resolution.go and is only applied later during finalization in phase_transitions.go.

As a result, verifiers with sufficient slot weight can submit `dealer_validity=false` together with an admissible complaint selector, push an honest dealer below quorum before the dispute is adjudicated, and cause that dealer's complaint to be discarded before the dealer has any chance to disprove it. Because the dealer's candidacy is never recomputed after complaint resolution, false complaint-driven votes can remain decisive, removing honest dealers from the final set or causing DKG failure due to insufficient candidate dealer slots.

Recommendation

Do not treat vote-based dealer candidacy as final before complaint resolution. Preserve all admitted complaints through the disputing phase, allow dealers to respond even if they initially miss vote-based candidacy, and only finalize dealer eligibility after complaint adjudication.

In practice, this means moving complaint filtering and quorum/failure checks until after dispute resolution, or recomputing dealer validity once false complaints have been disproven.

Alleviation

[Gonka, 05/13/2026]:
Issue acknowledged. Changes have been reflected in the commit `7d338a6d8834cbd27d73dfd9b05148e4dc683227` .

GEB-01 Discussion On Unpaired Burn/BurnFrom On Wrapped Supply In `wrapped-token` Contract

Category	Severity	Location	Status
Inconsistency	Informational	inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 100, 107	Acknowledged

Description

The `wrapped-token` contract explicitly forwards `Burn()` and `BurnFrom() to cw20-base. This means any holder (or approved spender) can destroy wrapped supply without emitting the bridge withdrawal message. If the intended bridge model assumes every burn corresponds to a withdrawal event, this creates a path where supply is reduced without triggering a release on the source chain. This may be acceptable by design (e.g., voluntary burn), but it should be documented and/or gated if unpaired burns are not desired.

Recommendation

The audit team would like to confirm with the team if this is the intended design.

Alleviation

[Gonka, 02/21/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> yes, this is the intended design and it will be reflected in the documentation

GEB-02 | Discussion On `UpdateMetadata` That Modifies Decimals

Category	Severity	Location	Status
Inconsistency	Informational	inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 68, 152~155	Acknowledged

Description

The `UpdateMetadata` message permits an account with the admin or creator role to modify the wrapped token's hardcoded decimal value 6.

`inference-chain/contracts/wrapped-token/src/contract.rs`

```
132 fn update_metadata(
133     deps: DepsMut,
134     info: MessageInfo,
135     name: String,
136     symbol: String,
137     decimals: u8,
138 ) -> Result<Response, ContractError> {
139     // Load both creator and admin addresses
140     let creator = CREATOR.load(deps.storage)?;
141     let admin = ADMIN.load(deps.storage)?;
142
143
// Allow both creator (inference module) and admin (governance module) to update
metadata

144     let is_creator = info.sender == creator;
145     let is_admin = info.sender == admin;
146
147     if !is_creator && !is_admin {
148         return Err(ContractError::Unauthorized {});
149     }
150
151
152     TOKEN_METADATA.save(
153         deps.storage,
154 @>      &TokenMetadataOverride { name: name.clone(), symbol: symbol.clone(),
 decimals },
155     )?;
156 //...
```

Recommendation

The audit team seeks clarification on this design choice, as a token's decimal configuration is generally expected to be

immutable.

Alleviation

[Gonka, 02/21/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> hardcoded decimal value 6 is not universal for any ERC-20 token. theoretically, any ERC-20 token can be bridged, in order to add support tokens, metadata should be added through the governance vote.

GEB-11 | `InstantiateMsg.marketing` Is Ignored

Category	Severity	Location	Status
Inconsistency	Informational	inference-chain/contracts/wrapped-token/src/contract.rs (82c43a4): 73~78	Resolved

Description

Within the `instantiate()`, the `marketing` is hardcoded, though the input `InstantiateMsg` contains the field `marketing`.

Recommendation

If the `marketing` is intended to be hardcoded, recommend removing `marketing` from `InstantiateMsg`.

Alleviation

[Gonka Team, 03/20/2026]:
Issue acknowledged. The team resolved the finding by removing the `marketing`. Changes have been reflected in the commit `8fbed2495ac6fe6b36aa234b92f24ca08243ebca`.

GEB-28 | Discussion On Donor Mechanism

Category	Severity	Location	Status
Design Issue	Informational	inference-chain/x/bls/keeper/dkg_initiation.go (82c43a4): 92	Resolved

Description

The protocol includes a donor mechanism designed to protect smaller validators by requiring larger validators to distribute some of their slots to smaller ones. In scenarios where one validator holds the vast majority of the total voting weight (e.g., a single validator with 99% and the remaining 99 validators sharing 1%), the slot allocation system (with 100 slots) results in each validator receiving an equal number of slots. This outcome can be perceived as inequitable to the validator with the highest weight.

While such an imbalance is unlikely to occur in practice, this mechanism could incentivize large validators to divide their stake among multiple smaller identities to maximize slot allocation, potentially undermining the intended fairness of the protocol. On the other hand, this mechanism prevent centralization of the threshold signing by allowing more small validators.

Recommendation

The audit team would like to understand the trade-off and confirm with the team if the current implementation aligns with the expected design.

Alleviation

[Gonka, 02/21/2026]:
We don't expect a single validator to have 99%. Our whole blockchain is built under the assumption of an honest majority (<1/3 power can be gained by attackers).

Given this assumption, do you see any attack vector when BLS can be compromised?

[CertiK, 02/21/2026]:
Thanks for clarifying your threat model. Under the honest-majority assumption, this finding is not a direct consensus-break issue; we classify it as a design-risk/tradeoff.

Our concern is that the donor mechanism can be gamed via stake splitting (Sybil-style identities): even without majority stake, a large operator may increase slot capture by distributing stake across multiple validators, while honest high-weight validators are diluted by forced slot redistribution. This does not necessarily violate your <1/3 assumption, but it can weaken slot-allocation fairness and may reduce threshold-signing robustness/liveness if donated slots concentrate on weaker operators.

So our understanding is: your intended design prioritizes decentralization/inclusion over strict stake-proportional slot fairness. Please confirm this is the expected behavior and accepted tradeoff.

[Gonka, 03/24/2026]:
this is a fair point about the tradeoff and prioritizing decentralization. this decision was made for the current state of the

network and is something that governance very realistically can change in the future.

At this moment the distribution of validators is such that if chain allocates slots without donor mechanism, we'd have very few validators for BLS.

Additionally, this attack vector "this mechanism could incentivize large validators to divide their stake among multiple smaller identities to maximize slot allocation" is not as risky as it might look since the validators for slot allocation are cut off at 100 top validators. So when splitting a large validator into smaller identities, you would still need for each of these identities to get into top 100 validators to get a single extra slot.

[Gonka, 04/08/2026]:
Issue acknowledged. After an internal discussion, the team decided to remove donor mechanism. Changes have been reflected in the commit `2e450beeb200611cd449119f560b06ae3c939a69` .

GEB-33 | Discussion On Old Epoch Keys Can Authorize Mint/Withdraw

Category	Severity	Location	Status
Design Issue	Informational	proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (82c43a4): 332, 393	Acknowledged

Description

In the current design, any stored epoch key can authorize withdrawal/mint operations.

Specifically, functions `withdraw` / `mintWithSignature` accept `cmd.epochId` without checking it's current. If an old validator set is compromised, the attacker can mint/withdraw arbitrary new requests with fresh `requestId` until that epoch key is deleted (365 days by default).

proposals/ethereum-bridge-contact/contracts/BridgeContract.sol

```
330     function withdraw(WithdrawalCommand calldata cmd)
external nonReentrant onlyNormalOperation {
331         // 1. Epoch Validation: Cache group key to avoid double SLOAD
332 @>      GroupKey memory groupKeyStruct = epochGroupKeys[cmd.epochId];
333         if (_isGroupKeyEmpty(groupKeyStruct)) {
334             revert InvalidEpoch();
335         }
```

```
391     function mintWithSignature(MintCommand calldata cmd)
external nonReentrant onlyNormalOperation {
392         // 1. Epoch Validation: Cache group key to avoid double SLOAD
393 @>      GroupKey memory groupKeyStruct = epochGroupKeys[cmd.epochId];
394         if (_isGroupKeyEmpty(groupKeyStruct)) {
395             revert InvalidEpoch();
396         }
```

Recommendation

The audit team would like to confirm with the team if this design is intended.

Alleviation

[Gonka, 02/21/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> It was done by design. We were trying to find balance in time period so that the user can 100% complete the transaction.

GEB-44 | Discussion On DKg's Status On COMPLETED And SIGNED

Category	Severity	Location	Status
Inconsistency	Informational	inference-chain/x/bls/keeper/threshold_signing.go (82c43a4): 24	Resolved

Description

The bridge currently relies on `COMPLETED` epochs, which only prove that the current DKG finished. Without requiring the `SIGNED` status (previous epoch attestation of the new group key), a compromised or newly captured cohort could rotate to a malicious key without prior-epoch endorsement. This preserves liveness but loses inter-epoch continuity guarantees. If the bridge is intended to trust a key lineage, it should gate on `SIGNED` and add a timeout/fallback so validation stalling cannot block progress.

Recommendation

The audit team would like to understand if this is an intended design that the bridge workflow only relies on `COMPLETED` status as the failure of transition from Completed to Signed could occur.

Alleviation

[Gonka, 02/21/2026]:
The bridge entirely relies on the signed status and won't accept a request until received signed new key. Not sure that we understand your concern. What would be the fallback that you recommend?

[CertiK, 02/21/2026]:
Thanks for the clarification. We believe there is a mismatch between intended behavior and current implementation.

In code, `RequestThresholdSignature()` currently allows both `COMPLETED` and `SIGNED` epochs, not `SIGNED-only`:

```
func (k Keeper) RequestThresholdSignature(ctx sdk.Context, signingData
types.SigningData) error {
    // Validate current epoch has completed DKG
    epochBLSData, found := k.GetEpochBLSData(ctx, signingData.CurrentEpochId)
    if !found {
        return fmt.Errorf("epoch BLS data not found for epoch %d",
signingData.CurrentEpochId)
    }

    // Verify epoch has completed DKG (has group public key)
@>  if epochBLSData.DkgPhase != types.DKGPhase_DKG_PHASE_COMPLETED &&
epochBLSData.DkgPhase != types.DKGPhase_DKG_PHASE_SIGNED {
        return fmt.Errorf("epoch %d DKG not completed, current phase: %s",
signingData.CurrentEpochId, epochBLSData.DkgPhase.String())
    }
//...
```

Our concern is continuity of trust across epochs:

- `COMPLETED` proves the new epoch DKG finished.
- `SIGNED` additionally proves prior-epoch attestation of the new key.

So if bridge requests are accepted at `COMPLETED`, the system is liveness-friendly but does not strictly enforce key-lineage continuity.

[Gonka, 04/08/2026]:

Issue acknowledged. The team resolved the finding by enforcing it to `SIGNED` status. Changes have been reflected in the commit `1164c17d53ecfb788f8f632b6c0cb528309ae459`.

GEB-45 Discussion On Slot Allocation When Participants Is More Than Slots

Category	Severity	Location	Status
Logical Issue	Informational	inference-chain/x/bls/keeper/dkg_initiation.go (82c43a4): 123, 147	Resolved

Description

When `nonZeroCount > totalSlots`, the code slices to the top `totalSlots` participants, so the participant count equals the slot count and all have non-zero weight.

inference-chain/x/bls/keeper/dkg_initiation.go, `AssignSlots()`

```
 122
// If we have more non-zero participants than slots, select the top N by weight
 123  @>  if nonZeroCount > int(totalSlots) {
 124          // Calculate weight of participants that will be excluded for logging
 125          excludedWeight := math.LegacyZeroDec()
 126          excludedCount := nonZeroCount - int(totalSlots)
 127
 128          // Sort by weight descending, then by address for determinism
 129          sort.Slice(sortedParticipants, func(i, j int) bool {
 130              if sortedParticipants[i].PercentageWeight.Equal(sortedParticipants[
j].PercentageWeight) {
 131                  return sortedParticipants[i].Address < sortedParticipants[j].
Address
 132              }
 133              return sortedParticipants[i].PercentageWeight.GT(sortedParticipants
[j].PercentageWeight)
 134          })
 135
 136          // Calculate weight of excluded participants (those beyond totalSlots)
 137          for i := int(totalSlots); i < len(sortedParticipants); i++ {
 138              excludedWeight = excludedWeight.Add(sortedParticipants[i].
PercentageWeight)
 139          }
 140
 141
// Defensive check: verify we can safely slice (should always be true given
nonZeroCount > totalSlots)

 142          if int(totalSlots) > len(sortedParticipants) {
 143              return nil, fmt.Errorf(
"internal error: totalSlots %d exceeds participant count %d", totalSlots, len(
sortedParticipants))
 144          }
 145
 146          // Keep only top totalSlots participants
 147  @>      sortedParticipants = sortedParticipants[:totalSlots]
```

Step 4 then forces “each non-zero participant gets at least one slot,” and Step 5 fixes the total to `totalSlots`. That combination forces the final allocation to be **exactly one slot per selected participant**, regardless of their relative weights. Heavy participants lose any extra slots they would have been assigned; the algorithm effectively becomes “pick the top N by weight, then give each exactly one slot,” dropping proportional weighting once oversubscribed.

```
230      // 4. Ensure every non-zero-weight participant has at least one slot.
231      for i, p := range sortedParticipants {
232          if p.PercentageWeight.IsZero() {
233              continue
234          }
235  @>      if assigned[i] > 0 {
236              continue
237          }
238
239          donor := findDonorIndex(assigned, remainders, sortedParticipants)
240          if donor == -1 {
241              return nil, fmt.Errorf(
"unable to assign at least one slot to participant %s", p.Address)
242          }
243
244          assigned[donor]--
245          assigned[i]++
246      }
```

```
248      // 5. Final validation: slot counts should sum to totalSlots.
249      checkTotal := int64(0)
250      for _, cnt := range assigned {
251          checkTotal += cnt
252      }
253  @>  if checkTotal != int64(totalSlots) {
254          return nil, fmt.Errorf("slot assignment mismatch: expected %d, got %d",
 totalSlots, checkTotal)
255      }
```

Recommendation

The audit team would like to confirm with the team if the current implementation aligns with the intended design.

Alleviation

[Gonka, 04/08/2026]:

Issue acknowledged. After an internal discussion, the team decided to remove donor mechanism. Changes have been reflected in the commit `2e450beeb200611cd449119f560b06ae3c939a69` .

GEB-46 | Discussion On Potential ETH/WGNK Address Collision

Category	Severity	Location	Status
Logical Issue	Informational	proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (82c43a4): 363~374	Resolved

Description

`withdraw()` handles `cmd.tokenContract == address(this)` as ETH,

`proposals/ethereum-bridge-contact/contracts/BridgeContract.sol`

```
363     if (cmd.tokenContract == address(this)) {
364         // ETH withdrawal: tokenContract == address(this) indicates ETH
365         require(address(this).balance >= cmd.amount, "Insufficient ETH balance"
);
366
367         // Use call{value:} for better gas compatibility (no 2300 gas limit)
368         (bool success, ) = cmd.recipient.call{value: cmd.amount}("");
369         require(success, "ETH transfer failed");
370     } else {
371         // ERC-20 withdrawal: standard token transfer
372         IERC20(cmd.tokenContract).safeTransfer(cmd.recipient, cmd.amount);
373     }
```

however `address(this)` is also the address of WGNK ERC-20, as documented and implemented at the beginning of the contract

```
10 /**
11  * @title BridgeContract
12
 * @dev Ethereum bridge contract and WGNK (Wrapped Gonka) ERC-20 token with BLS
threshold signatures
13
 * @notice This contract serves as both the bridge and the WGNK token, enabling
seamless cross-chain transfers
14  */
15 contract BridgeContract is ERC20, Ownable, ReentrancyGuard {...
```

If WGNK ERC-20 (`address(this)`) is registered on the onchain, a signed withdrawal for WGNK will wrongly attempt to send ETH, likely failing or draining any ETH, conversely, ETH withdrawals are indistinguishable from WGNK token withdrawals.

Recommendation

The audit team would like to confirm with the team if `address(this)` will be reserved for ETH.

Alleviation

[Gonka, 04/07/2026]:
Yes, `address(this)` is reserved for ETH in `withdraw()` and is not a bug for the current architecture, for the following structural reasons:

`WGNK` is the native asset, not a wrapped token. The `BridgeContractAddresses` registry holds `address(BridgeContract)` as the sentinel for the native GONKA <> ETH bridge. On the Inference Chain side, `IsBridgeContractAddress` catches this and routes to `HandleNativeTokenRelease` rather than ever creating a CW20 representation of `WGNK`. So `WGNK` (address(this)) is never registered as a CW20 wrapped token contract.

The withdrawal signing payload on Gonka for `WGNK` mints is generated in `prepareBridgeMintSignatureData`, which hardcodes `address(BridgeContract)` as the bridge sentinel, matching `address(this)` in `withdraw()` on Ethereum. This is intentional and consistent. The relayer correctly classifies WGNK burns as native burns (using the contract \in bridgeAddrSet AND to == 0x0 rule), and Gonka routes them to escrow release, not wrapped token minting.

Still on our end we add protection during new bridge address registration from a possibility that address was already registered as wrapped token address: https://github.com/gonka-ai/gonka/pull/1022

GEB-50 Unused Function `computeParticipantPublicKey()` In `bls_crypto.go`

Category	Severity	Location	Status
Coding Style, Volatile Code	Informational	inference-chain/x/bls/keeper/bls_crypto.go (82c43a4): 14~48	Resolved

Description

The function `computeParticipantPublicKey()` is never used in current codebase.

`inference-chain/x/bls/keeper/bls_crypto.go`

```
14
// computeParticipantPublicKey computes individual BLS public key for participant's slots
15 func (k Keeper) computeParticipantPublicKey(epochBLSData *types.EpochBLSData, slotIndices []uint32) ([]byte, error) {
16     // Initialize aggregated public key as G2 identity
17     var aggregatedPubKey bls12381.G2Affine
18     aggregatedPubKey.SetInfinity()
19
20     // For each slot assigned to this participant
21     for _, slotIndex := range slotIndices {
22         // For each valid dealer's commitments
23         for dealerIdx, isValid := range epochBLSData.ValidDealers {
24             if !isValid || dealerIdx >= len(epochBLSData.DealerParts) {
25                 continue
26             }
27
28             dealerPart := epochBLSData.DealerParts[dealerIdx]
29             if dealerPart == nil || len(dealerPart.Commitments) == 0 {
30                 continue
31             }
32
33             // Evaluate dealer's commitment polynomial at this slot index
34             // This requires polynomial evaluation using the commitments
35             slotPublicKey, err := k.evaluateCommitmentPolynomial(dealerPart.Commitments, slotIndex)
36             if err != nil {
37                 return nil, fmt.Errorf("failed to evaluate commitment polynomial for dealer %d slot %d: %w", dealerIdx, slotIndex, err)
38             }
39
40             // Add to aggregated public key
41             aggregatedPubKey.Add(&aggregatedPubKey, &slotPublicKey)
42         }
43     }
44
45     // Return compressed public key bytes
46     pubKeyBytes := aggregatedPubKey.Bytes()
47     return pubKeyBytes[:], nil
48 }
```

Recommendation

Recommend removing unused function.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by removing the unused `computeParticipantPublicKey()` path from

active code and disabling the associated timing test that referenced it. Changes have been reflected in the commit `79e2d1c17a7ce535fd58e3662c3c8724bb755b11` .

GEB-51 Discussion On Post-Processing Of Failed Threshold Signing Requests

Category	Severity	Location	Status
Design Issue	Informational	inference-chain/x/bls/keeper/threshold_signing.go (82c43a4): 360	Resolved

Description

In the current codebase, in case that a request of threshold signing fails, it only invokes `emitThresholdSigningFailed()` but does not have refund or rollback mechanism.

Recommendation

The audit team would like to understand how the post-process works in case of the threshold signing request fails.

Alleviation

[Gonka Team, 04/14/2026]:
Issue acknowledged. The team resolved the finding by adding explicit post-processing for failed bridge-related threshold signing requests. Failed or expired bridge signing requests now persist pending refund context, invoke inference-module BLS hooks, and automatically refund escrowed native tokens or re-mint burned wrapped tokens as appropriate. The patch also adds bounded auto-retry and manual/governance cancellation paths where explicit recovery is needed. Changes have been reflected in the commit `2535320e26313d7f04df4eeaea76f4b0d206f2c0` .

GEB-52 | Discussion On EVM Monitor And Transaction Submission

Category	Severity	Location	Status
Design Issue	Informational		Acknowledged

Description

In the current codebase, the off-chain component does not have the EVM monitoring services, transaction submission and the service of bridge transaction submission to Gonka chain.

Recommendation

The audit team would like to understand how they are designed to complete the bridge workflow.

Alleviation

[Gonka, 03/17/2026]:
The team acknowledged the issue and decided not to implement the recommended change in the current engagement, providing the following statement:

> we are using the copy of the original etherium bridge client with light sync version on top of it: we don't do verification and treat it as unnecessary as we send only safe blocks. https://github.com/gonka-ai/bridge-geth/tree/bridge

GEB-58 `ProcessThresholdSigningRequested()` Incorrectly Returns Error

Category	Severity	Location	Status
Coding Issue	Informational	threshold_signing.go: 61~70	Resolved

Description

The function `ProcessThresholdSigningRequested` intends to skip if the node is not a participant. However, it first calls `GetOrRecoverVerificationResult`, which will return error when the node is not a participant.

- **decentralized-api/internal/bls/threshold_signing.go**:

```
  61 @>  result, err := bm.GetOrRecoverVerificationResult(epochId)
  62     if err != nil {
  63         logging.Warn(thresholdSigningLogTag+"Failed to get verification result"
, inferenceTypes.BLS, "epoch_id", epochId, "error", err)
  64         return fmt.Errorf("failed to get verification result: %w", err)
  65     }
  66
  67 @>  if !result.IsParticipant {
  68         logging.Debug(thresholdSigningLogTag+
"Not a participant in this epoch, skipping", inferenceTypes.BLS, "epoch_id", epochId
)
  69         return nil
  70     }
```

- **decentralized-api/internal/bls/manager.go**:

```
 176         completed, err := bm.setupAndPerformVerification(epochID, &res.
EpochData)
 177         if err != nil {
 178             return nil, fmt.Errorf("failed to recover: %w", err)
 179         }
 180 @>        if !completed {
 181             return nil, fmt.Errorf("not a participant in epoch %d", epochID)
 182         }
```

Recommendation

Recommend ensuring the error treatment consistent.

Alleviation

[Gonka Team, 03/20/2026]:

Issue acknowledged. The team resolved the finding by changing `GetOrRecoverVerificationResult()` so that “not a participant” is returned as a normal verification result (`IsParticipant = false`) rather than as an error, and by ensuring threshold-signing processing explicitly skips non-participants. Changes have been reflected in the commit `52c2b687c0a3231644458a7f50d14ea28e28d283`.

GEB-63 Discussion On Missing Emergency Pause Across Bridge Execution Paths

Category	Severity	Location	Status
Volatile Code	Informational	proposals/ethereum-bridge-contact/contracts/BridgeContract.sol (04/21-6d0dee1): 15	Acknowledged

Description

The bridge lacks a fast emergency circuit breaker across its active execution paths. On Solidity, `withdraw()` and `mintWithSignature()` are gated by `onlyNormalOperation`, but the contract only leaves normal operation through the 30-day timeout path. This means the Ethereum-side owner/admin cannot immediately halt bridge execution if valid but incorrect signed messages start appearing.

The Cosmos/WASM wrapped-token contract similarly lacks a bridge-specific pause. Its Withdraw path can continue burning wrapped tokens and creating outbound bridge requests, while module-driven Mint can continue minting wrapped tokens for completed inbound bridge events.

This does not by itself bypass BLS verification or create unauthorized withdrawals. However, if the source-chain bridge logic, relayer flow, validator threshold, deployment configuration, or epoch-key management fails, the absence of a coordinated pause increases incident blast radius and complicates recovery.

Recommendation

The audit team would like to check with the team if missing pause/unpause mechanism is intended or it should be added for defense-in-depth.

Alleviation

[Gonka, 05/22/2026]:
This is intentional. Bridge security comes from the validator BLS threshold, not from a privileged operator. A fast pause would hand the multisig a unilateral freeze over user funds, creating risks of centralized control and a single point of failure.

We acknowledge the tradeoff and plan to add a recovery mechanism that addresses this without reintroducing centralized control in the future.

APPENDIX | GONKA - ETHEREUM BRIDGE

Audit Scope

gonka-ai/gonka
inference-chain/contracts/wrapped-token/src/contract.rs
proposals/ethereum-bridge-contact/contracts/BridgeContract.sol
inference-chain/contracts/community-sale/src/contract.rs
inference-chain/x/bls/keeper/bls_crypto.go
inference-chain/x/bls/keeper/dkg_initiation.go
inference-chain/x/bls/keeper/msg_server_dealer.go
inference-chain/x/bls/keeper/msg_server_group_validation.go
inference-chain/x/bls/keeper/msg_server_threshold_signing.go
inference-chain/x/bls/keeper/msg_server_verifier.go
inference-chain/x/bls/keeper/phase_transitions.go
inference-chain/x/bls/keeper/threshold_signing.go
inference-chain/x/bls/module/genesis.go
inference-chain/x/bls/types/genesis.go
inference-chain/x/bls/types/message_request_threshold_signature.go
inference-chain/x/bls/types/message_submit_dealer_part.go
inference-chain/x/bls/types/message_submit_group_key_validation_signature.go
inference-chain/x/bls/types/message_submit_partial_signature.go
inference-chain/x/bls/types/message_submit_verification_vector.go
inference-chain/x/inference/keeper/bridge_native.go

gonka-ai/gonka
inference-chain/x/inference/keeper/bridge_transaction.go
inference-chain/x/inference/keeper/bridge_utils.go
inference-chain/x/inference/keeper/bridge_wrapped_token.go
inference-chain/x/inference/keeper/msg_server_register_bridge_addresses.go
inference-chain/x/inference/keeper/msg_server_request_bridge_mint.go
inference-chain/x/inference/keeper/msg_server_request_bridge_withdrawal.go
inference-chain/x/inference/types/message_request_bridge_withdrawal.go
decentralized-api/internal/bls/dealer.go
decentralized-api/internal/bls/verifier.go
inference-chain/contracts/wrapped-token/src/error.rs
inference-chain/contracts/wrapped-token/src/lib.rs
inference-chain/contracts/wrapped-token/src/msg.rs
inference-chain/contracts/wrapped-token/src/state.rs
inference-chain/contracts/community-sale/src/error.rs
inference-chain/contracts/community-sale/src/lib.rs
inference-chain/contracts/community-sale/src/msg.rs
inference-chain/contracts/community-sale/src/state.rs
inference-chain/x/bls/keeper/keeper.go
inference-chain/x/bls/keeper/msg_server.go
inference-chain/x/bls/keeper/msg_update_params.go
inference-chain/x/bls/keeper/params.go
inference-chain/x/bls/keeper/query.go

gonka-ai/gonka
inference-chain/x/bls/keeper/query_epoch_data.go
inference-chain/x/bls/keeper/query_params.go
inference-chain/x/bls/keeper/query_threshold_signing.go
inference-chain/x/bls/module/autocli.go
inference-chain/x/bls/module/module.go
inference-chain/x/bls/types/codec.go
inference-chain/x/bls/types/errors.go
inference-chain/x/bls/types/expected_keepers.go
inference-chain/x/bls/types/keys.go
inference-chain/x/bls/types/msg_update_params.go
inference-chain/x/bls/types/params.go
inference-chain/x/bls/types/types.go
inference-chain/x/inference/keeper/bridge.go
inference-chain/x/inference/keeper/msg_register_wrapped_token_contract.go
inference-chain/x/inference/keeper/query_bridge.go
inference-chain/x/inference/keeper/query_bridge_token.go
inference-chain/x/inference/keeper/query_bridge_transaction.go
inference-chain/x/inference/keeper/query_wrapped_token_balances.go
inference-chain/x/inference/types/message_request_bridge_mint.go
decentralized-api/internal/bls/group_validation.go
decentralized-api/internal/bls/manager.go
decentralized-api/internal/bls/threshold_signing.go

Finding Categories

Categories	Description
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
	No description available.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.