// Security Assessment

08.27.2025 - 09.08.2025

Bridge
Mezo

HALBORN

Bridge - Mezo

Prepared by: HALBORN

Last Updated 09/18/2025

Date of Engagement: August 27th, 2025 - September 8th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	0	0	2	3	4

TABLE OF CONTENTS

1. Introduction

`Halborn` was engaged by `Mezo` to perform a security assessment of the bridge implementation. The assessment period began on August 27, 2025, and concluded on September 8, 2025. The assessment was scoped to the codebase supplied to `Halborn`; commit hashes and additional details are provided in the *Scope* section of this report.

The engagement broadly covered the Mezo bridge implementation components described in the provided context: the Solidity contract **MezoBridgeV2**, and the Go bridge components located at **/precompile/assetsbridge**, **/x/bridge**, and the sidecar at **/ethereum/sidecar**. Bridge-in functionality was excluded from the scope, as it had previously been audited by another company and evaluated in a public contest. The objective of the engagement was to identify security weaknesses and to provide concrete remediation guidance for the Mezo bridge-out implementation and the associated validator and sidecar logic.

2. Assessment Summary

Nine days were allocated for this engagement and one full-time security engineer was assigned to review the security of the repositories in scope. The assigned engineer possessed deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objectives of this assessment were to:

- Identify potential security vulnerabilities within the `smart contracts` and `CosmosSDK` project.
- Verify that the bridge functions as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were mostly acknowledged by the `Mezo team`. The primary recommendations were:

- `Validator identifiers should be stable, or attestation bitmaps should be cleaned up on membership changes, to prevent double voting.`
- `Gas refunds in attestBridgeOut should only be issued for valid attestations, with refundable deposits, stake deductions, and rate limiting to prevent reimbursement pool draining.`
- `Panic conditions should be replaced with structured error handling, invariant checks, and governed recovery paths instead of chain-halting panics.`
- `Token enabling should be atomic or gated on a non-zero minimum, with fee calculations adjusted to avoid zero results from rounding.`
- `Validators should check token enablement on-chain and perform pre-send simulations with retry limits to prevent infinite loops and resource exhaustion.`
- `Sidecar validator shuffling should rely on atomic snapshots of validator sets to avoid race conditions and inconsistent queueing.`
- `Withdrawal fees should be locked per commitment at bridge time and compared against a user-specified maximum to prevent fee manipulation.`
- `Pending entries should be automatically re-evaluated after validator removals so that those meeting the new threshold complete without manual intervention.`
- `Administrative functions should emit indexed events with old and new values to improve observability and auditability.`

3. Test Approach And Methodology

A layered testing strategy was applied, combining code review, design analysis, and operational testing techniques. The sequence of phases described in the supplied materials included research and scoping (repository and RFC review), manual code review of critical paths in both `Solidity` and `Go` sources, analysis of failure modes and edge cases, and review of sidecar submission logic. Effort was weighted toward manual review for protocol and consensus logic (validator management, attestations, economic flows), while automated scans were relied upon for standard checks; on-chain tests were described conceptually for validation of attestation and token-enable interactions. The principal focus was placed on canonical invariants (one-vote-per-validator, fee preservation, safe retry behavior) and on the interaction surface between on-chain contracts and off-chain sidecars and validators.

Coverage was intended to be thorough for bridge-out related paths: attestation lifecycle (`attestBridgeOut`, `attestBridgeOutWithSignatures`), validator management (`addBridgeValidator`, `removeBridgeValidator`), withdrawal completion (`validateAssetsUnlocked`, `withdrawBTC`), token mapping and minimums (`createERC20TokenMapping`, `setMinBridgeOutAmount`), and sidecar submission queue behavior. Emphasis was placed on design-level vulnerabilities that could permit economic abuse, liveness failures, or chain-halting conditions.

4. RISK METHODOLOGY

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

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

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

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

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

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

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

ATTACK COMPLEXITY (AX):

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

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

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

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

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

INTEGRITY (I):

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

AVAILABILITY (A):

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

DEPOSIT (D):

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

YIELD (Y):

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

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

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

Impact I is calculated using the following formula:

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

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

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

SCOPE (S):

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

METRICS:

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

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: mezod

(b) Assessed Commit ID: 1ea1913

(c) Items in scope:

- x/bridge/module.go
- x/bridge/types/errors.go
- x/bridge/types/keys.go
- x/bridge/types/assets_unlocked.go
- x/bridge/types/assets_unlocked_test.go
- x/bridge/types/params.go
- x/bridge/types/query.pb.go
- x/bridge/types/query.pb.gw.go
- x/bridge/types/genesis.go
- x/bridge/types/genesis.pb.go
- x/bridge/types/genesis_test.go
- x/bridge/types/interfaces.go
- x/bridge/types/assets_locked.go
- x/bridge/types/assets_locked_test.go
- x/bridge/types/bridge.pb.go
- x/bridge/types/erc20.go
- x/bridge/types/erc20_test.go
- x/bridge/keeper/outflow_limit_test.go
- x/bridge/keeper/pause.go
- x/bridge/keeper/pause_test.go
- x/bridge/keeper/abci_test.go
- x/bridge/keeper/assets_unlocked.go
- x/bridge/keeper/assets_unlocked_test.go
- x/bridge/keeper/outflow_limit.go
- x/bridge/keeper/abci.go
- x/bridge/keeper/genesis.go
- x/bridge/keeper/keeper.go
- x/bridge/keeper/params.go
- x/bridge/keeper/query_server.go
- x/bridge/keeper/erc20.go
- x/bridge/keeper/btc.go
- x/bridge/keeper/assets_locked.go
- x/bridge/abci/vote_extension.go
- x/bridge/abci/vote_extension_test.go
- x/bridge/abci/preblock_test.go
- x/bridge/abci/proposal.go

- x/bridge/abci/proposal_test.go
- x/bridge/abci/interfaces.go
- x/bridge/abci/preblock.go
- x/bridge/abci/types/vote_extension.pb.go
- x/bridge/abci/types/proposal.pb.go
- x/bridge/client/cli/query.go
- precompile/assetsbridge/pause_test.go
- precompile/assetsbridge/setup_test.go
- precompile/assetsbridge/byte_code.go
- precompile/assetsbridge/min_amount.go
- precompile/assetsbridge/min_amount_test.go
- precompile/assetsbridge/outflow_limit.go
- precompile/assetsbridge/outflow_limit_test.go
- precompile/assetsbridge/pause.go
- precompile/assetsbridge/abi.json
- precompile/assetsbridge/assets_bridge.go
- precompile/assetsbridge/bridge_out.go
- precompile/assetsbridge/bridge_out_test.go
- precompile/assetsbridge/IAssetsBridge.sol
- precompile/assetsbridge/observability.go
- precompile/assetsbridge/sequence_tip.go
- precompile/assetsbridge/btc.go
- precompile/assetsbridge/erc20.go
- ethereum/sidecar/mock_contracts_test.go
- ethereum/sidecar/server.go
- ethereum/sidecar/server_test.go
- ethereum/sidecar/submission_queue.go
- ethereum/sidecar/submission_queue_test.go
- ethereum/sidecar/contracts.go
- ethereum/sidecar/contracts_test.go
- ethereum/sidecar/mock_bridge_worker_test.go
- ethereum/sidecar/assets_unlocked_test.go
- ethereum/sidecar/attestation_validator.go
- ethereum/sidecar/attestation_validator_test.go
- ethereum/sidecar/batch_attestation.go
- ethereum/sidecar/batch_attestation_test.go
- ethereum/sidecar/chain_test.go
- ethereum/sidecar/assets_unlocked.go
- ethereum/sidecar/client.go
- ethereum/sidecar/client_mock.go
- ethereum/sidecar/cli/ethereum_sidecar.go
- ethereum/sidecar/cli/flags.go
- ethereum/sidecar/mezotime/mezotime.go
- ethereum/sidecar/types/ethereum_sidecar.pb.go

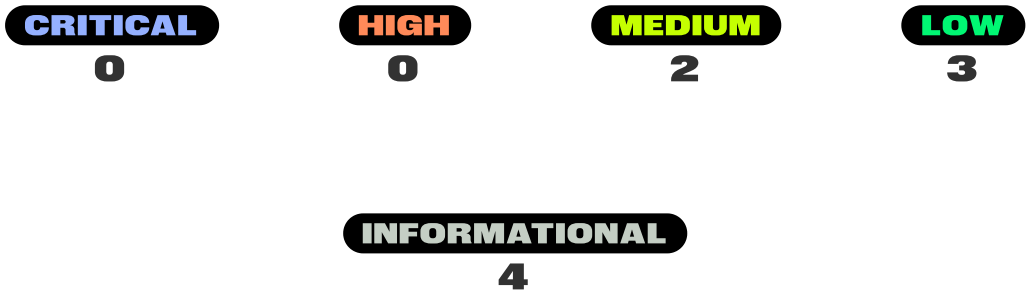
Out-of-Scope: External dependencies and economic attacks.

REMEDIATION COMMIT ID:

- a957af7
- a957af7

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DOUBLE VOTING VULNERABILITY THROUGH VALIDATOR ID REASSIGNMENT	MEDIUM	PARTIALLY SOLVED - 09/02/2025
REIMBURSEMENT POOL DRAIN VIA ATTESTATION SPAM	MEDIUM	RISK ACCEPTED - 09/17/2025
PANIC CONDITIONS IN BRIDGE MODULE	LOW	RISK ACCEPTED - 09/17/2025
NON ATOMICITY TOKEN ENABLING ALLOWS FEE BYPASS	LOW	RISK ACCEPTED - 09/17/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
RESOURCE EXHAUSTION VIA DISABLED ERC20 TOKEN BRIDGE OUT	LOW	RISK ACCEPTED - 09/17/2025
RACE CONDITION IN SIDECAR VALIDATOR SHUFFLING	INFORMATIONAL	ACKNOWLEDGED - 09/17/2025
FEE UPDATE AFTER BRIDGE COMMITMENT	INFORMATIONAL	ACKNOWLEDGED - 09/17/2025
THRESHOLD STUCK ENTRIES AFTER VALIDATOR REMOVAL	INFORMATIONAL	PARTIALLY SOLVED - 09/02/2025
MISSING EVENTS FOR ADMINISTRATIVE FUNCTIONS	INFORMATIONAL	ACKNOWLEDGED - 09/17/2025

7. FINDINGS & TECH DETAILS

7.1 DOUBLE VOTING VULNERABILITY THROUGH VALIDATOR ID REASSIGNMENT

// MEDIUM

Description

The MezoBridge contract permits a single validator to cast multiple votes on the same withdrawal entry. When validators are removed, validator IDs are reassigned but attestation bitmaps are left unchanged, allowing remaining validators to attest again for entries on which they have already voted.

The core issue is that **a single validator can vote multiple times on the same withdrawal entry**, violating the bridge's fundamental security assumption:

```
// When a validator is removed, IDs are reassigned but bitmaps remain unchanged
if (validatorIndex != lastValidatorIndex) {
    address lastValidator = bridgeValidators[lastValidatorIndex];
    bridgeValidators[validatorIndex] = lastValidator;
    bridgeValidatorIDs[lastValidator] = validatorID; // ID reassignment without clearing bitmaps
}
```

Attack Scenario:

1. **Initial State**: ValidatorD (ID=4) attests to Entry2, setting bit 4 in the attestation bitmap.
2. **Validator Removal**: ValidatorB (ID=2) is removed from the validator set.
3. **ID Reassignment**: ValidatorD is reassigned from ID=4 to ID=2.
4. **Double Vote**: ValidatorD is able to attest to Entry2 again using the new ID=2, setting bit 2.
5. **Result**: Entry2 records two attestations while only a single physical validator has participated.

Impact:

- **Bridge Threshold Bypass**: Required attestation counts can be reached with fewer distinct validators than intended.
- **Consensus Manipulation**: The security model, which assumes one vote per validator per entry, is undermined.

Proof of Concept

The proof of concept is as follows:

```
import { ethers, helpers } from "hardhat"
import { loadFixture } from "@nomicfoundation/hardhat-toolbox/network-helpers"
import { expect } from "chai"
import { HardhatEthersSigner } from "@nomicfoundation/hardhat-ethers/signers"
import deployPortal from "./fixtures/deployPortal"
import { MezoBridge } from "../typechain"

const { createSnapshot, restoreSnapshot } = helpers.snapshot

// Chain enum matching the MezoBridge contract
enum Chain {
  ETHEREUM = 0,
  BITCOIN = 1,
}

```

```typescript
// AssetsUnlocked struct matching the contract
interface AssetsUnlocked {
  unlockSequenceNumber: bigint
  recipient: string  // hex string for bytes type
  token: string
  amount: bigint
  chain: number
}

describe("MEZO-01 PoC: Validator ID Reassignment Bug", () => {
  let mezoBridge: MezoBridge
  let deployer: HardhatEthersSigner
  let validatorA: HardhatEthersSigner
  let validatorB: HardhatEthersSigner
  let validatorC: HardhatEthersSigner
  let validatorD: HardhatEthersSigner
  let mockERC20: any

  before(async () => {
    // Use the existing deployment fixture which properly sets up MezoBridge
    const fixtures = await loadFixture(deployPortal)

    mezoBridge = fixtures.mezoBridge
    deployer = fixtures.deployer
    validatorA = fixtures.validatorOne
    validatorB = fixtures.validatorTwo
    validatorC = fixtures.validatorThree
    validatorD = fixtures.validatorFour
    mockERC20 = fixtures.USDC  // Use the USDC mock from fixtures

    // Enable the mock ERC20 token for bridging
    await mezoBridge.connect(deployer).enableERC20Token(await mockERC20.getAddress(), 1n)

    // Add 4 validators
    await mezoBridge.connect(deployer).addBridgeValidator(validatorA.address) // ID = 1
    await mezoBridge.connect(deployer).addBridgeValidator(validatorB.address) // ID = 2
    await mezoBridge.connect(deployer).addBridgeValidator(validatorC.address) // ID = 3
    await mezoBridge.connect(deployer).addBridgeValidator(validatorD.address) // ID = 4
  })

  describe("Validator ID reassignment causes attestation vote inheritance and valid orphan attes
    let entry1Hash: string
    let entry2Hash: string
    let entry1: AssetsUnlocked
    let entry2: AssetsUnlocked

    before(async () => {
      await createSnapshot()

      // Fund the MezoBridge with tokens so it can process withdrawals
      const bridgeAddress = await mezoBridge.getAddress()
      await mockERC20.connect(deployer).mint(bridgeAddress, ethers.parseEther("1000"))

      // Create two different withdrawal entries
      const Recipient1 = ethers.hexlify(ethers.randomBytes(20))
      const Recipient2 = ethers.hexlify(ethers.randomBytes(20))

      entry1 = {
        unlockSequenceNumber: 1n,
        recipient: Recipient1,
        token: await mockERC20.getAddress(),
        amount: 1000n,
        chain: Chain.ETHEREUM,
      }

      entry2 = {
        unlockSequenceNumber: 2n,
        recipient: Recipient2,
        token: await mockERC20.getAddress(),
        amount: 2000n,
        chain: Chain.ETHEREUM,
      }

      // Calculate entry hashes
      entry1Hash = ethers.keccak256(
        ethers.AbiCoder.defaultAbiCoder().encode(
          ["tuple(uint256,bytes,address,uint256,uint8)"],
          [[entry1.unlockSequenceNumber, entry1.recipient, entry1.token, entry1.amount, entry1.c
```

```
        )
      )

      entry2Hash = ethers.keccak256(
        ethers.AbiCoder.defaultAbiCoder().encode(
          ["tuple(uint256,bytes,address,uint256,uint8)"],
          [[entry2.unlockSequenceNumber, entry2.recipient, entry2.token, entry2.amount, entry2.c
        )
      )

      // ValidatorB attests to entry1
      await mezoBridge.connect(validatorB).attestBridgeOut(entry1)

      // ValidatorD attests to entry2
      await mezoBridge.connect(validatorD).attestBridgeOut(entry2)

      // Remove ValidatorB - this causes ValidatorD to move to ID=2
      await mezoBridge.connect(deployer).removeBridgeValidator(validatorB.address)
    })

    after(async () => {
      await restoreSnapshot()
    })

    it("should show ValidatorD inherits ValidatorB's attestation for entry1, attestation for ent
      // Check validator IDs after removal
      expect(await mezoBridge.bridgeValidatorIDs(validatorB.address)).to.equal(0) // ValidatorB
      expect(await mezoBridge.bridgeValidatorIDs(validatorD.address)).to.equal(2) // ValidatorD

      // Check entry1 attestations (ValidatorB voted on this)
      const entry1Attestation = await mezoBridge.attestations(entry1Hash)
      expect((entry1Attestation >> 2n) & 1n).to.equal(1n) // Bit 2 set - ValidatorD inherited th

      // Check entry2 attestations (ValidatorD voted on this)
      const entry2Attestation = await mezoBridge.attestations(entry2Hash)
      expect((entry2Attestation >> 4n) & 1n).to.equal(1n) // Bit 4 still set but ValidatorD no l
      expect((entry2Attestation >> 2n) & 1n).to.equal(0n) // Bit 2 not set - ValidatorD's new po

      // ValidatorD can now vote AGAIN on entry2 with their new ID!
      await mezoBridge.connect(validatorD).attestBridgeOut(entry2)

      const newEntry2Attestation = await mezoBridge.attestations(entry2Hash)
      expect((newEntry2Attestation >> 4n) & 1n).to.equal(1n) // Bit 4 still set (orphaned from V
      expect((newEntry2Attestation >> 2n) & 1n).to.equal(1n) // Bit 2 now set - ValidatorD voted
      // If we then add a new validator he will inherit ValidatorD's old ID's attestations
    })
  })
})
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (6.7)

Recommendation

Consider implementing a comprehensive attestation cleanup mechanism when validator membership changes to maintain system integrity. Upon validator removal, the system could automatically clear any orphaned attestation bits from the affected validator's previous ID and provide a grace period for legitimate pending entries to be re-evaluated.

Remediation Comment

PARTIALLY SOLVED: The **Mezo team** partially remediated this issue by implementing a two-step validator removal process and will **create a public document** for bridge management processes. The issue remains as governance-level mitigation rather than code fix.

Remediation Hash

a957af79623005d817adb811899a3fc799ba386b

7.2 REIMBURSEMENT POOL DRAIN VIA ATTESTATION SPAM

// MEDIUM

Description

A malicious validator can drain the ReimbursementPool by repeatedly calling `attestBridgeOut` with different entries. Each call triggers a gas refund regardless of whether the attestation is valid or reaches the threshold, enabling profitable pool drainage.

As a result validators can systematically drain the reimbursement pool.

Code Location

In `MezoBridge.sol`, the `attestBridgeOut` function provides gas refunds for all attempts: :

```
function attestBridgeOut(AssetsUnlocked calldata entry)
    external nonReentrant refundable(Refund(msg.sender, false, 1000)) {
}
```

Proof of Concept

The proof of concept is as follows:

```
import { ethers } from "hardhat"
import { loadFixture } from "@nomicfoundation/hardhat-toolbox/network-helpers"
import deployPortal from "./fixtures/deployPortal"

describe("Minimal Drain Test", () => {
  enum Chain { ETHEREUM = 0 }

  it("should show validator profit from reimbursement pool drain", async () => {
    const { mezoBridge, reimbursementPool, validatorOne, validatorTwo, validatorThree, deployer,

    // Setup 3 validators (threshold = 3)
    await mezoBridge.connect(deployer).addBridgeValidator(validatorOne.address)
    await mezoBridge.connect(deployer).addBridgeValidator(validatorTwo.address)
    await mezoBridge.connect(deployer).addBridgeValidator(validatorThree.address)
    await mezoBridge.connect(deployer).updateReimbursementPool(await reimbursementPool.getAddres
    await reimbursementPool.connect(governance).authorize(await mezoBridge.getAddress())

    // Fund reimbursement pool
    await deployer.sendTransaction({
      to: await reimbursementPool.getAddress(),
      value: ethers.parseEther("1"),
    })

    // Check balances before
    const initialValidatorBalance = await ethers.provider.getBalance(validatorOne.address)
    console.log("Validator balance before:", ethers.formatEther(initialValidatorBalance), "ETH")

    const entry = {
      unlockSequenceNumber: 1,
      recipient: ethers.zeroPadValue("0x01", 20),
      token: await mezoBridge.tbtcToken(),
      amount: 1000000n,
      chain: Chain.ETHEREUM,
    }

    await mezoBridge.connect(validatorOne).attestBridgeOut(entry)

    // Check balances after
    const finalValidatorBalance = await ethers.provider.getBalance(validatorOne.address)
    console.log("Validator balance after:", ethers.formatEther(finalValidatorBalance), "ETH")
```

```
    const profit = finalValidatorBalance - initialValidatorBalance
    console.log("Validator profit:", ethers.formatEther(profit), "ETH")
  })
})
```

BVSS

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

Recommendation

The `attestBridgeOut` function should be modified so that gas refunds are issued only when an attestation is accepted (for example, verified and having reached the required threshold). When an attestation is not accepted, a refundable deposit should be required, or the refunded amount should be deducted from the validator's stake. Additionally, per-validator rate limiting and a per-block call cap should be implemented to prevent attestation spam from draining the `ReimbursementPool`.

Remediation Comment

RISK ACCEPTED: The **Mezo team** accepted the risk of this finding and stated the following:

> *We acknowledge the issue but decide to leave it unaddressed - the complexity and the gas cost of remediation strongly outweigh the benefits. Attestations are submitted asynchronously, so introducing deferred refunds requires an additional mechanism to manage them efficiently. The issue can be mitigated on the operational level. First, the reimbursement pool is funded in small batches. Second, during the PoA phase, misbehaving validators can be easily detected and quickly removed by the governance. The attack does not lead to any profits - the goal of the refunds is making the transaction cost-neutral, so this attack is actually pure griefing. Moreover, the individual attestation mode is the fallback to the batch attestation mode and should be rarely used in real-world scenarios.*

7.3 PANIC CONDITIONS IN BRIDGE MODULE

// LOW

Description

Multiple panic conditions are present in the bridge module and in the sidecar. These panic points could cause chain execution to halt if triggered by corrupted state or unanticipated edge cases. If any panic condition is triggered, manual intervention is required to recover.

These panics appear to have been implemented as intentional circuit breakers to prevent further state corruption. This design choice is potentially hazardous because any unexpected edge case can halt the entire chain, and no graceful degradation or automated recovery mechanisms are provided.

Code Location

1. **Unmarshaling panics** in `keeper/assets_unlocked.go`:

```
// Lines 25, 44, 56, 79, 209, 223
err := sequenceTip.Unmarshal(bz)
if err != nil {
    panic(err)  // Could trigger on corrupted state
}
```

2. **Supply invariant panic** in `keeper/abci.go`:

```
if !minted.Sub(burnt).Equal(supply) {
    panic(fmt.Sprintf("supply of BTC is not balanced: minted %s, burnt %s",
          minted.String(), burnt.String()))
}
```

3. **Invalid token address panic** in `keeper/erc20.go`:

```
mezoToken, err := evmtypes.HexAddressToBytes(v.MezoToken)
if err != nil {
    panic(fmt.Sprintf("invalid mezo token address in state %v: %v",
          v.MezoToken, err))
}
```

4. **Consensus violation panic** in `abci/proposal.go`:

```
mezoToken, err := evmtypes.HexAddressToBytes(v.MezoToken)
if err != nil {
    panic(fmt.Sprintf("invalid mezo token address in state %v: %v",
          v.MezoToken, err))
}
```

5. **Sidecar initialization panics** in `ethereum/sidecar/server.go`:

```
// Network connection failure (line 207)
panic(fmt.Sprintf("failed to connect to the Ethereum network: %v", err))

// Contract initialization failure (line 213)
panic(fmt.Sprintf("failed to initialize MezoBridge contract: %v", err))
```

```
697
698 // Validator ID lookup failure (line 298)
    panic(fmt.Sprintf("failed to get bridge validator ID: %v", err))
```

6. **Sidecar signing panic** in `ethereum/sidecar/batch_attestation.go`:

```
103 // Key signing failure
104 panic(fmt.Sprintf("unable to sign batch attestation payload: %v", err))
```

BVSS

AO:A/AC:L/AX:H/R:N/S:C/C:N/A:C/I:N/D:N/Y:N (4.1)

Recommendation

All `panic()` calls should be removed and replaced with explicit error returns and centralized invariant handling. Consensus-critical checks must be enforced via the `module invariant registry` using non-panicking handlers that emit alerts and trigger governed recovery or safe block rejection. Unmarshalling failures must be treated as recoverable errors, with defined validation or migration paths and events to support manual remediation. Sidecar initialization and signing errors must be handled using structured error values, retry/backoff policies, health checks, and supervised restart mechanisms instead of panics.

Remediation Comment

RISK ACCEPTED: The **Mezo team** accepted the risk of this finding and stated the following:

> *All panic conditions are intentional design decisions for fail-fast behavior and state protection. No remediation needed as behavior is by design.*

7.4 NON ATOMICITY TOKEN ENABLING ALLOWS FEE BYPASS

// LOW

Description

Token enabling requires two separate transactions on Mezo, which creates a potential window that allows withdrawal fees to be bypassed due to a zero minimum bridge amount and integer-division rounding effects.

Token enabling on Mezo is performed in two transactions:

```
// Transaction 1: Enable token mapping
assetsBridge.createERC20TokenMapping(sourceToken, mezoToken)

// Transaction 2: Set minimum amount
assetsBridge.setMinBridgeOutAmount(mezoToken, minAmount)
```

Between these transactions, a zero minimum is in effect and bridging is permitted without the intended minimum threshold, enabling fee bypasses. While exploitation is likely unprofitable in many cases because of Mezo-side transaction costs, the issue becomes more significant as fee levels decrease. The primary impacts are:

- Temporary bypass of withdrawal fees during token setup
- Inconsistent fee collection for small transfer amounts
- Protocol revenue loss during the race-condition window

Attack Scenario

1. **Transaction 1 submitted**: `createERC20TokenMapping(TOKEN_A, MEZO_TOKEN_A)`
2. **Race-condition window**: The token is bridgeable with a zero minimum
3. **Attacker exploits**: Multiple `bridgeOut` calls with small amounts are executed and fees are calculated as zero
4. **Transaction 2 submitted**: `setMinBridgeOutAmount(MEZO_TOKEN_A, properMinimum)`

The attack is constrained by several limitations:

- Mezo gas must be paid for each `bridgeOut` transaction
- Mezo transaction costs will often exceed the fee savings on the originating chain
- Real token balances on Mezo are required to perform the withdrawals
- Exploitation is limited to the narrow time window between the two transactions

Code Location

In `mezod/x/bridge/keeper/assets_unlocked.go`, the minimum defaults to zero when no value is set:

```
if len(bz) == 0 {
    return math.ZeroInt()  // Returns 0 if no minimum set
}
```

In `MezoBridge.sol`, the Ethereum-side fee calculation uses integer division that rounds down to zero for sufficiently small amounts:

```
609 uint256 feeAmount = (amount * withdrawalFee) / 10000;
610
611 // Bypass threshold formula: floor((10000 / withdrawalFee) - 1)
612 // Examples:
613 // 0.01 fee (1 bps):   amounts ≤9999 wei bypass fees
614 // 0.1% fee (10 bps):  amounts ≤999 wei bypass fees
615 // 1% fee (100 bps):   amounts ≤99 wei bypass fees
616 // 10% fee (1000 bps): amounts ≤9 wei bypass fees
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (3.4)

Recommendation

Ensure token enabling is performed atomically by requiring the minimum bridge amount to be set in the same transaction that creates the `ERC20` mapping or by blocking `bridgeOut` until a non-zero minimum is stored (i.e., reject bridge operations when `min == 0`). Additionally, the fee calculation should be adjusted to avoid zero results from integer division by rounding up or enforcing a minimum fee (for example, `feeAmount = max((amount * withdrawalFee + 10000 - 1) / 10000, 1)`).

Remediation Comment

RISK ACCEPTED: The **Mezo team** accepted the risk of this finding and stated the following:

> *We acknowledge that, but the actual impact is negligible. First, the governance is typically a SAFE account and does transactions like such in batches. We will also create a public document describing the bridge management processes to guide the governance (https://github.com/mezo-org/mezod/issues/570) around actions like this. Last but not least, even if the issue occurs, the amounts that allow bypassing the fee must be very small, so the amount of potentially lost fees is quite insignificant.*

7.5 RESOURCE EXHAUSTION VIA DISABLED ERC20 TOKEN BRIDGE OUT

// LOW

Description

When an ERC20 token is enabled for bridge-out on Mezo but is not enabled on the Ethereum MezoBridge contract, validators are caused to enter infinite retry loops that generate continuous failed Ethereum transactions, resulting in severe resource exhaustion.

Attack Flow

1. **Bridge Out Succeeds on Mezo**: A token mapping is present, funds are burned, and an `AssetsUnlocked` event is emitted.
2. **Attestation Fails on Ethereum**: The `validateAssetsUnlocked()` call reverts with `InvalidToken(token)`.
3. **Infinite Retry Loop**: Validators retry every 10 seconds without termination.

Impact Analysis

Potential Resource Exhaustion Scenarios

Scenario A: Transaction Simulation (Lower Impact)

If go-ethereum's `Transact()` method performs a pre-send simulation:

- **Gas Estimation Failure**: Transaction construction is expected to fail due to the predicted revert.
- **No Network Transactions**: Failed attempts are not propagated to the Ethereum network.
- **CPU/Memory Impact**: Continuous local simulation attempts occur every 10 seconds.
- **Cost**: Minimal, limited to local computation.

Scenario B: Actual Transaction Submission (Higher Impact)

If transactions are submitted without pre-simulation:

- **Continuous Failed Transactions**: A new Ethereum transaction is created every 10 seconds per validator for each stuck sequence.
- **Gas Burn Rate**: Approximately 0.0006 ETH per attempt, equating to roughly 5.18 ETH/day per validator per stuck sequence.
- **Blockchain Bloat**: Thousands of failed transactions are permanently recorded on-chain.
- **Network Congestion**: Failed transactions compete with legitimate traffic, increasing congestion and latency.

Code Location

In `bridge_out.go`:

```
0 | // Only checks local mapping existence
1 | if _, ok := GetERC20TokenMappingFromMezoToken(token); !ok {
2 |     return fmt.Errorf("unsupported token")
3 | }
```

In `MezoBridge.sol`:

```
485 | // Checks if token is enabled on Ethereum
486 | if (entry.token != _tbtcToken && ERC20Tokens[entry.token] == 0) {
487 |     revert InvalidToken(entry.token);
488 | }
```

In `server.go`:

```
1107 | for i := 0; ; i++ {  // Infinite loop - no max attempts
1108 |     // 10 second backoff
1109 |     if i > 0 {
1110 |         time.After(attestationProcessBackoff)  // 10 seconds
1111 |     }
1112 |
1113 |     tx, err := s.bridgeContract.AttestBridgeOut(bridgeAssetsUnlocked)
1114 |     if err != nil {
1115 |         continue  // Retry forever
1116 |     }
1117 |     break
1118 | }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (3.4)

Recommendation

The `Mezo` bridge-out flow must be modified so that token enablement is verified against the on-chain `MezoBridge ERC20Tokens` mapping (or a trusted on-chain view) prior to burning or emitting `AssetsUnlocked`. Validators must be modified to perform a pre-send `eth_call`/`estimateGas` check and to cease retrying after a configurable maximum number of attempts (or to mark the sequence as permanently stuck and raise an alert), in order to prevent infinite failed transactions and resource exhaustion.

Remediation Comment

RISK ACCEPTED: The **Mezo team** accepted the risk of this finding and stated the following:

> *We acknowledge, but this is again strongly dependent on the proper governance processes. We will provide a public document describing the bridge management to guide the governance (https://github.com/mezo-org/mezod/issues/570) and prevent issues like this. Moreover, we consider the part about resource exhaustion as potentially invalid. Indeed, the validators will retry indefinitely for the problematic request (which is intended and aims to make issues like this immediately visible from the monitoring system), but the contract binding implemented in the Ethereum sidecar and used to submit transactions always does the gas estimation prior to the actual transaction. That said, validators wouldn't burn actual funds for failing transactions.*

7.6 RACE CONDITION IN SIDECAR VALIDATOR SHUFFLING

// INFORMATIONAL

Description

A race condition is present in the sidecar submission-queue mechanism when validators are added or removed while attestation processing is ongoing. This causes sidecar shuffles to be computed against different validator-array sizes, producing conflicting submission timings.

Impact

- **Queue collision**: Concurrent submissions by multiple validators are enabled (intended staggering is negated).
- **Missing attestations**: Incorrect submission delays may be calculated for some validators, causing attestations to be omitted or submitted outside expected windows.
- **Inconsistent behavior**: Different sidecars may operate under different validator-set assumptions, producing non-deterministic submission ordering across nodes.

Attack Scenario

1. **Fallback mode is triggered**: A batch attestation attempt fails and the system falls back to individual submissions.
2. **Sidecar A queries**: `BridgeValidatorsCount() = 5`.
3. **An admin adds a validator**: A new validator is registered on the bridge.
4. **Sidecar B queries**: `BridgeValidatorsCount() = 6`.
5. **Different shuffles are computed**:
- Sidecar A shuffles `[1,2,3,4,5]` using the same seed.
- Sidecar B shuffles `[1,2,3,4,5,6]` using the same seed.
6. **Result**: A single validator is assigned different queue positions by different sidecars, producing submission conflicts.

Code Location

In `submission_queue.go`, a deterministic shuffle is used by the sidecar to stagger validator attestation submissions:

```
52  func (s *submissionQueue) calculateSubmissionQueue(sequenceNumber *big.Int) ([]uint8, error) {
53      validatorCount, err := s.bridgeContract.BridgeValidatorsCount() // <- Race point
54
55      validatorIDs := make([]uint8, count)
56      for i := uint64(0); i < count; i++ {
57          validatorIDs[i] = uint8(i + 1)
58      }
59
60      seed := sequenceNumber.Int64()
61      return s.shuffleValidatorIDs(validatorIDs, seed), nil
62  }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

A validator shuffle must be computed from a canonical, atomic snapshot of validator IDs obtained via a single read (call `BridgeValidatorsCount` and fetch all IDs once). That snapshot must be used for shuffling. The read must be protected either by a local mutex or transactional lock, or by including and validating an on-chain `epoch`/`version` (re-check the count/epoch and abort or retry if changed). This prevents inconsistent shuffles when validators are added or removed.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** acknowledged this issue and stated the following:

> *We acknowledge this finding, but the impact is negligible. Two validators may submit an attestation at the same time, worst case scenario is that a validator will submit a surplus attestation, which does not even lead to any cost on the validators end as a proper transaction will be reimbursed from the pool.*

7.7 FEE UPDATE AFTER BRIDGE COMMITMENT

// INFORMATIONAL

Description

When assets are bridged from Mezo to Ethereum, the withdrawal fee displayed at bridge time is not locked in. The fee is instead sampled at withdrawal time. This permits the owner to increase the withdrawal fee after users have locked their funds, which forces users to either accept the higher fee or forfeit timely recovery of their assets.

Attack Scenario

1. A user bridges 1 BTC from Mezo when `withdrawalFee = 100` (1%).
2. Validators attest to the withdrawal entry.
3. The owner calls `updateWithdrawalFee(500)`, increasing the fee to 5%.
4. The user calls `withdrawBTC()` expecting to be charged a 0.01 BTC fee.
5. The user is instead charged a 0.05 BTC fee (five times the expected amount).

Impact

- **Broken user expectations**: The fee displayed at bridge time may differ from the fee actually charged at withdrawal.
- **No protection mechanism**: No mechanism is provided to set a maximum acceptable fee or to cancel the withdrawal if the fee increases.
- **Forced acceptance**: Recovery of funds is conditioned on acceptance of any fee in effect at withdrawal time.

Code Location

```
// Fee is determined at withdrawal time, not bridge time
function _collectWithdrawalFee(address token, uint256 amount) internal {
    uint256 _withdrawalFee = withdrawalFee;  // Current fee, not historical
    uint256 feeAmount = (amount * _withdrawalFee) / BASIS_POINTS_DENOMINATOR;
    // ...
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (1.0)

Recommendation

The withdrawal fee should be recorded at bridge time and persisted per withdrawal commitment (for example, `mapping(bytes32 => uint256) commitmentFee`) so that the withdrawal logic uses the stored historical fee rather than the mutable global `withdrawalFee`. An optional `maxFee` parameter should be accepted by the `withdraw` function and compared to the stored fee so that the call is automatically

reverted if the fee exceeds the user's expected cap. Events should be emitted for fee changes and for commitments.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** acknowledged this issue and stated the following:

> *We acknowledge this finding. This can be mitigated on the governance level by limiting any fee changes to periods without active bridge-out requests. In practice, this fee will rarely change. Last but not least, we plan to introduce a flat fee and modify the fee mechanism so we will remove the current percentage fee and eliminate this issue completely.*

7.8 THRESHOLD STUCK ENTRIES AFTER VALIDATOR REMOVAL

// INFORMATIONAL

Description

A stuck-entry condition is observed when validators are removed after a partial set of attestations. Entries that already hold a sufficient number of attestations for the reduced threshold can remain uncompleted because an automatic completion path is not implemented. Manual completion is required by calling `attestBridgeOutWithSignatures`.

Scenario

1. **Initial setup**: Eleven validators are configured (threshold = 8).
2. **Partial attestation**: Validators 1–7 attest entry E (7/8 is therefore insufficient).
3. **Validator removal**: Validators 8–11 are removed.
4. **New state**: Seven validators remain (threshold = 5).
5. **Result**: Entry E contains seven attestations but remains stuck (7 ≥ 5 yet no automatic completion is triggered).

Impact

- **User funds are rendered unclaimable**: Legitimate withdrawals are prevented from being processed automatically.
- **Manual intervention is required**: Off-chain signature collection must be performed to call `attestBridgeOutWithSignatures`, or an additional validator must be added so that `attestBridgeOut` can be called to complete the entry.

Code Location

The `attestBridgeOut` function performs the threshold check only after a new attestation is added:

```
function attestBridgeOut(AssetsUnlocked calldata entry) external {
    // Add new attestation
    attestations[attestationKey] = updatedBitmap;

    // Only check completion after new attestation
    if (_countSetBits(updatedBitmap) < attestationThreshold()) {
        return; // Exit without completing
    }

    // Complete the entry
    _withdraw(entry);
}
```

Proof of Concept

The proof of concept is as follows:

```
describe("Threshold Stuck Entries", () => {
  it("demonstrates entry stuck after validator removal", async () => {
    // Setup 11 validators
    for (let i = 0; i < 11; i++) {

```

```
    await mezoBridge.addBridgeValidator(validators[i].address);
  }

  // Threshold = 8, validators 1-7 attest (insufficient)
  for (let i = 0; i < 7; i++) {
    await mezoBridge.connect(validators[i]).attestBridgeOut(entry);
  }

  // Remove validators 8-11 (threshold drops to 5)
  for (let i = 7; i < 11; i++) {
    await mezoBridge.removeBridgeValidator(validators[i].address);
  }

  // Entry has 7 attestations ≥ 5 threshold but stuck
  const attestations = await mezoBridge.attestationsCount(entryHash);
  const threshold = await mezoBridge.attestationThreshold();
  const completed = await mezoBridge.confirmedUnlocks(entry.unlockSequenceNumber);

  console.log(`Attestations: ${attestations}, Threshold: ${threshold}, Completed: ${completed}`);
  // Output: Attestations: 7, Threshold: 5, Completed: false

  // All remaining validators already attested - cannot trigger completion
  expect(attestations).to.be.gte(threshold);
  expect(completed).to.be.false; // Entry permanently stuck
 });
});
```

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (1.0)

Recommendation

Ensure that **validator removal triggers a re-evaluation of all pending entries** so that any entry with a sufficient number of existing attestations relative to the new threshold is automatically completed without requiring manual off-chain intervention. This can be achieved by maintaining per-entry attestation counts or enqueuing entries for re-check when validator sets change, ensuring that legitimate withdrawals are not stuck indefinitely and that system state remains consistent even after validator membership updates.

Remediation Comment

PARTIALLY SOLVED: The **Mezo team** partially remediated this issue by implementing a two-step validator removal process. Manual unblocking possible through `attestBridgeOutWithSignatures` function.

Remediation Hash

a957af79623005d817adb811899a3fc799ba386b

7.9 MISSING EVENTS FOR ADMINISTRATIVE FUNCTIONS

// INFORMATIONAL

Description

Administrative functions in the bridge precompile don't emit events, reducing observability for monitoring and auditing.

Affected Functions

Functions without events:

- `SetPauserMethod` - Changes pauser address silently
- `PauseBridgeOutMethod` - Pauses bridge without notification
- `SetOutflowLimitMethod` - Updates limits without trace

Functions with proper events (for comparison):

- `SetMinBridgeOutAmountMethod` ✅
- `CreateERC20TokenMappingMethod` ✅
- `BridgeOutMethod` ✅

BVSS

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

Recommendation

Events should be emitted for all administrative, state-changing functions (e.g., `SetPauserMethod`, `PauseBridgeOutMethod`, `SetOutflowLimitMethod`). Event parameters should be indexed for affected addresses and should include explicit fields for previous and new values, plus a timestamp or block number, to ensure off-chain observability and auditability.

Remediation Comment

ACKNOWLEDGED: The **Mezo team** acknowledged this issue and stated the following:

> *We acknowledge this finding. We consider adding the proposed events as part of https://github.com/mezo-org/mezod/issues/571. Doing it is a breaking upgrade and requires a chain halt, so we will try to pack it with one of the future upgrades.*
