Mezo

Security Assessment

March 18th, 2025 — Prepared by OtterSec

Yordan Stoychev anatomic@osec.io

James Wang james.wang@osec.io

Table of Contents

© 2025 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Thesis engaged OtterSec to assess the `mezo` and `mezoBridge` programs. This assessment was conducted between February 28th and March 11th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a vulnerability where there is a potential denial-of-service risk in the BTC bridging logic, where bridging BTC to a blocked address will result in the coin-sending process failing (OS-MZO-ADV-00).

We also made recommendations to ensure accurate estimates of the fees on the Bitcoin network before bridging BTC to tokenized-BTC (OS-MZO-SUG-00), and suggested implementing a proper database for the sidecar and prune only events that have been processed (OS-MZO-SUG-02). We further advised considering cases where benign validators are allowed to reject proposals and vote extensions from other benign validators (OS-MZO-SUG-04).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/thesis/mezo-portal. This audit was performed against commit 9b657da and e35350c.

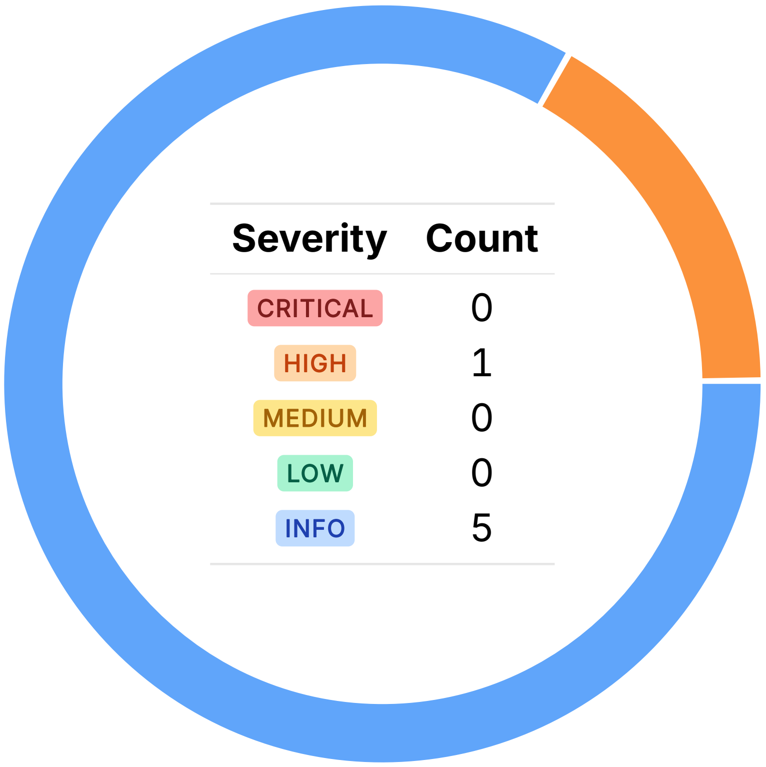
A brief description of the programs is as follows:

Name	Description
mezo	Mezo chain is a blockchain that utilizes Bitcoin as its base token while supporting EVM compatibility. It enables the transfer of Bitcoin to Mezo through a tBTC (tokenized bitcoin) bridge, which initially operates on Ethereum-to-Mezo bridging. Additionally, the same bridging mechanism allows ERC-20 tokens to be transferred from Ethereum to Mezo, facilitating asset interoperability.
mezoBridge	The bridge's pillar on Ethereum is implemented as a Solidity contract called MezoBridge. This contract holds TBTC and arbitrary ERC20 tokens in custody and emits AssetsLocked events, which are monitored by the Ethereum sidecar.

03 — Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	0
LOW	0
INFO	5

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-MZO-ADV-00	HIGH	RESOLVED ✓	There is a potential denial-of-service risk in the `BTC` bridging logic, where bridging `BTC` to a blocked address will result in `SendCoinsFromModuleToAccount` failing.

DOS Due to Failure to Send Coins HIGH OS-MZO-ADV-00

Description

The `BTC` bridging logic is vulnerable to denial-of-service. Specifically, the issue arises from the possibility of bridging `BTC` to a `BlockedAddr`, resulting in the failure of `SendCoinsFromModuleToAccount` in `assets_locked::mintBTC`. An attacker may exploit the system by sending BTC bridging requests to an address that is known to be blocked or restricted, resulting in the minting and transfer process failing.

>_ *mezod/x/bridge/keeper/assets_locked.go* GO

```go
func (k Keeper) mintBTC(
	ctx sdk.Context,
	recipient sdk.AccAddress,
	amount math.Int,
) error {
	[...]
	// Send the minted coins from x/bridge module account to the final recipient.
	err = k.bankKeeper.SendCoinsFromModuleToAccount(
		ctx,
		types.ModuleName,
		recipient,
		coins,
	)
	if err != nil {
		return fmt.Errorf("failed to send coins: %w", err)
	}
	[...]
}
```

Remediation

Verify if the recipient address is in the blocked list or subject to restrictions before attempting to send coins. This will prevent the failure of `SendCoinsFromModuleToAccount`.

Patch

Resolved in PR#430.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-MZO-SUG-00	The `tBTC` amount returned is an approximation that normally underestimates the bridged value. However, if the actual fees are lower than the maximum assumed, it may slightly overestimate `tBTC`, potentially resulting in the transfer of more tokens than are available.
OS-MZO-SUG-01	`ExportGenesis` does not persist privilege-related information, potentially resulting in the loss of access control settings upon chain restart.
OS-MZO-SUG-02	The current caching approach is insufficient when dealing with high volumes, resulting in the bridge falling behind and potentially pruning non-processed events.
OS-MZO-SUG-03	We recommended caching `req.MaxTxBytes` in `PrepareProposalHandler` to avoid relying on its potentially modified value by sub-handlers, ensuring consistent block size checks.
OS-MZO-SUG-04	The ABCI threat model may not fully account for cases where benign validators reject proposals or vote extensions from other benign validators. Issues arise when proposal handling logic rejects valid proposals due to size constraints or failed transaction injections.

Overestimation of tBTC Amount

OS-MZO-SUG-00

Description

`BitcoinBridge::finalizeBTCBridging` calls `_finalizeDeposit`, which calculates the `tbtcAmount` utilizing fee parameters that are not known exactly at finalization. Instead, it subtracts the maximum allowed fees (for optimistic minting and Bitcoin transaction fees) from the deposit amount. Although this approach usually underestimates the amount of `tBTC` to be minted (providing a safety margin), there may be cases where actual fees are lower than these maximums, resulting in the function overestimating the minted `tBTC`.

>_ *solidity/contracts/BitcoinBridge.sol* SOLIDITY

```solidity
function finalizeBTCBridging(
    uint256 btcDepositKey,
    address recipient
) external {
    [...]
    (
        uint256 initialDepositAmount,
        uint256 tbtcAmount,
        bytes32 expectedExtraData
    ) = _finalizeDeposit(btcDepositKey);
    [...]
    _bridge(recipient, tbtcToken, tbtcAmount);
}
```

Remediation

Ensure accurate estimates of the fees on the Bitcoin network before bridging `BTC` to `tBTC`.

Patch

The Mezo team is aware of this risk. After careful evaluation, the chance and magnitude of fee shortage are considered low enough, and the Mezo team will fund the shortages shall it arise.

© 2025 Otter Audits LLC. All Rights Reserved.

Lack of Persistence of Privilege Information OS-MZO-SUG-01

Description

`genesis::ExportGenesis` is responsible for exporting the state of the PoA (Proof-of-Authority) module to be reloaded later via `InitGenesis`. However, it does not export privilege-related information. When the blockchain is restarted utilizing a genesis file generated from `ExportGenesis`, the privileged roles will not be restored. Consequently, this results in governance disruptions, unauthorized actions, or even a loss of administrative control, breaking the integrity of the protocol.

>_ *mezod/x/poa/keeper/genesis.go* GO

```go
// ExportGenesis writes the current store values
// to a genesis file, which can be imported again
// with InitGenesis
func (k Keeper) ExportGenesis(ctx sdk.Context) *types.GenesisState {
	return &types.GenesisState{
		Params:     k.GetParams(ctx),
		Owner:      k.GetOwner(ctx).String(),
		Validators: k.GetAllValidators(ctx),
	}
}
```

Remediation

Modify `ExportGenesis` to include privilege-related data to ensure proper restoration of privileges.

Patch

The Mezo team acknowledges the finding, but decides to keep the original implementation, since privilege roles can be restored by the chain admin upon restart, and the code will be deprecated in a transitions from PoA to PoS in the near future.

Bottlenecks in Event Processing OS-MZO-SUG-02

Description

A cache is utilized to temporarily store events, with a limit of 500,000 events. This cache is designed to hold events that the system will process before they are discarded or committed to long-term storage. Mezo allows only 10 events per block, and with an average Ethereum block time of 12 seconds, this results in approximately 120 events every 12 seconds. This is significantly lower than the event rate Ethereum may produce. As a result, if the bridge is frequently used, Mezo may eventually fall behind the `sidecar`, resulting in non-processed events being pruned.

>_ *ethereum/sidecar/server.go* SOLIDITY

```
var (
    [...]
    // cachedEventsLimit is a number of events to keep in the cache.
    // Size of Sequence: 32 bytes
    // Size of Recipient: 42 bytes
    // Size of Amount: 32 bytes
    // Struct overhead and padding: ~16bytes
    // Total size of one event: 122 bytes (0.12KB)
    // Assuming we want to allocate ~64MB for the cache, we can store ~546k events.
    // For simplicity, let's make 500k events our limit. Even if deposits hit 1000
    // daily mark, that would give 50 days of cached events which should be more than
    // enough.
    cachedEventsLimit = 500000
)
```

Remediation

Implement a proper database for the `sidecar` and prune only events that have been processed. While this adds some complexity to the design, it would greatly enhance the system’s robustness.

Patch

Mezo team is aware of the limitation, but decides to avoid any premature optimizations since performance tests indicate that the current implementation is sufficient to handle the expected workload. Should congestion occur, the Mezo team will bump up the cache limit in the short term and start developing database support for long-term robustness.

Proper Handling of Dynamic Block Size OS-MZO-SUG-03

Description

In `proposal::PrepareProposalHandler`, the loop iterates over the `draftTxs` and checks whether adding a transaction will exceed the `req.MaxTxBytes` (the maximum allowed block size for transactions). However, `req.MaxTxBytes` is not a constant value during the execution of the handler because sub-handlers may modify this value. Specifically, the connect call path (which is not enabled in Mezo) may change `req.MaxTxBytes`.

>_ *mezod/app/abci/proposal.go* SOLIDITY

```
func (ph *ProposalHandler) PrepareProposalHandler() sdk.PrepareProposalHandler {
    return func(
            ctx sdk.Context,
            req *cmtabci.RequestPrepareProposal,
    ) (*cmtabci.ResponsePrepareProposal, error) {
        for _, tx := range draftTxs {
            txLen := int64(len(tx))
            if txsBytes+txLen > req.MaxTxBytes {
                    break
            }
            txs = append(txs, tx)
            txsBytes += txLen
        }
        [...]
    }
}
```

Remediation

cache `req.MaxTxSize` before the loop starts. This ensures that the same block size limit is utilized throughout the entire process, regardless of whether any sub-handler modifies it during execution.

Patch

The Mezo team acknowledged this finding, but decides to keep the original implementation since the re-counted amount of bytes are small and should not have any materialistic impact of chain throughput, and fixes will increase the complexity of the code base.

Benign Proposal Rejection OS-MZO-SUG-04

Description

There are potentail mismatches between `PrepareProposalHandler` and `ProcessProposalHandler` in the ABCI framework that may result in the rejection og benign validator proposals. If an injected transaction (`injectedTx`) exceeds `req.MaxTxSize`, the validator may propose an empty proposal. However, `ProcessProposalHandler` currently has a hardcoded rejection for empty proposals (as shown below). This implies that a benign validator (who correctly rejected an oversized `injectedTx`) may have their proposal incorrectly rejected simply because it is empty.

>_ *mezod/app/abci/proposal.go* SOLIDITY

```
func (ph *ProposalHandler) ProcessProposalHandler() sdk.ProcessProposalHandler {
    return func(
            ctx sdk.Context,
            req *cmtabci.RequestProcessProposal,
    ) (*cmtabci.ResponseProcessProposal, error) {
        [...]
        if len(req.Txs) == 0 {
            [...]
            // Note that if the app-level pseudo-tx was not injected but there
            // are regular transactions in the proposal, we will fail in the
            // next condition where we try to unmarshal the first transaction
            // as an injected pseudo-tx. A regular transaction will cause
            // unmarshalling failure and, we will reject the proposal as well.
            // This is not the most elegant way to handle this case but probably
            // the only one that is possible here.
            return nil, fmt.Errorf("empty proposal")
        }
        var injectedTx types.InjectedTx
        if err := injectedTx.Unmarshal(req.Txs[0]); err != nil {
                // If unmarshaling fails, we cannot recover, so return an error.
                return nil, fmt.Errorf("failed to unmarshal injected tx: %w", err)
        }
        [...]
    }
}
```

Also, if all subhandlers fail, the validator gives up on injection and proposes an empty transaction set. However, `ProcessProposalHandler` may reject the proposal if it contains an empty transaction list (as shown above). Thus, a validator that follows protocol may have its proposal rejected, even though it did nothing malicious.

Remediation

Ensure to consider cases where benign validators are allowed to reject proposals and vote extensions from other benign validators.

Patch

THe Mezo team is aware of this and considers it an inherent part of the system design. Errors such as subhandler failures are expected to not happen under normal execution, and in cases mentioned above, a chain halt is desired to allow manual investigation to resolve underlying issues.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2025 Otter Audits LLC. All Rights Reserved.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2025 Otter Audits LLC. All Rights Reserved.