MANTLE NETWORK

Mantle L2 Rollup

Security Assessment Report

Version: 2.2

July, 2023

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Mantle Network's smart contracts and related Golang software. For both areas, the review focused solely on the security aspects of the implementation, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contracts or Golang code. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Mantle Network's smart contracts and Golang code contained within the scope of the security review.

A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation.

Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Mantle Network's smart contracts and Golang code.

Overview

Mantle is an EVM-compatible Layer 2 (L2) Optimistic Rollup network designed to utilise the strong security guarantees of Ethereum while reducing its cost and latency. Building on the framework designed by Optimism, another Layer 2 Optimistic Rollup, Mantle seeks to reduce transaction costs further by integrating the use of EigenLayer for Data Availability (DA) without compromising on security.

In addition, Mantle has expanded the verification systems to include Multi-Party Computation (MPC) in order to minimize trust assumptions of sequencer behaviour and reduce the time delay necessary for withdrawals from Mantle L2. Mantle uses their own token BIT (Proposed to transition to the name MNT prior to Mantle's launch) for gas payments as well as being staked as a deposit for actors involved in the MPC and DA systems.

Security Assessment Summary

This timeboxed security review was conducted on the files hosted on the Mantle Network's repository and were assessed at commit 3e2b6db.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying issues/vulnerabilities associated with the business logic implementation of the components in scope.

For smart contracts this includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

For the Golang libraries and modules, this includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime and use of the Ethereum protocol.

To support this review, the testing team used the following automated testing tools:

For Solidity based smart contracts:

- Mythril: `https://github.com/ConsenSys/mythril`
- Surya: `https://github.com/ConsenSys/surya`

For Golang code:

- `golangci-lint`: `https://github.com/golangci/golangci-lint`
- `semgrep-go`: `https://github.com/dgryski/semgrep-go`
- `go-geiger`: `https://github.com/jlauinger/go-geiger`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 38 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 5 issues.
- Medium: 8 issues.
- Low: 16 issues.
- Informational: 8 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Mantle Network's smart contracts and Golang code. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
MNT-01	Elected TSS Nodes Can Act Without Any Deposit	Critical	Resolved
MNT-02	No Mechanism To Flag Sequencer Fraud	High	Closed
MNT-03	Default Values & Out-of-Bounds Panics In Merkle Tree Implementation	High	Resolved
MNT-04	Elected TSS Nodes Can Avoid Slashing By Having Insufficient Deposits	High	Resolved
MNT-05	TSS Nodes Set Includes Slashed Node By Default	High	Closed
MNT-06	Precompiled Contract Not Updated	High	Closed
MNT-07	L2Geth Client Private Key Stored Without Encryption	Medium	Closed
MNT-08	Lack Of Signature Size Verification Checks	Medium	Resolved
MNT-09	Incorrect Error Returned With Empty Signature	Medium	Resolved
MNT-10	Lack Of Array Size Checks Before Slicing Or Referencing Array Elements	Medium	Resolved
MNT-11	Lack Of Size Checks In `createUUID()` Function	Medium	Resolved
MNT-12	Lack Of Size Validation Of Byte Arrays Used With BLS	Medium	Closed
MNT-13	TSS Nodes Reporting Slashing Are Vulnerable To Front Running	Medium	Resolved
MNT-14	TSS Manager Is A Single Point Of Failure	Medium	Closed
MNT-15	Sensitive Information Passed In Command Line Arguments	Low	Closed
MNT-16	Sequencer Address Updates Can Immobilise Contract	Low	Resolved
MNT-17	TSS Node Design Uncertainty	Low	Resolved
MNT-18	If TSS Nodes Fall Below Quorum Size No TSS Decisions Can Be Made	Low	Resolved
MNT-19	TSS Nodes Could Be Paid The Same Fee Twice	Low	Resolved
MNT-20	TSS Nodes Can Report Themselves For Slashing To Reduce Penalty	Low	Resolved
MNT-21	Hardcoded Addresses Of `BIT` Token	Low	Resolved
MNT-22	Lack Of Upper Bounds Checks On `fraudProofPeriod`	Low	Closed
MNT-23	Gas Exhaustion In `resetRollupBatchData()` Loop	Low	Closed
MNT-24	Unreachable Error Handling In Proxyd Cache	Low	Resolved
MNT-25	Mnemonic Cannot Be Used Without Supplying Private Key	Low	Resolved
MNT-26	Nil Pointer Panic If DTL Returns Malformed Data	Low	Resolved

MNT-27	Unhandled Panic If Metadata Is Set To Nil	Low	Resolved
MNT-28	Lack of Error Handling in `Hex2Bytes()`	Low	Closed
MNT-29	Incorrect Usage Of Context In `NewDriver()` & `NewDataService()`	Low	Closed
MNT-30	Waitgroup Counter Not Incremented	Low	Resolved
MNT-31	`hashAfterPops()` Uses Incorrect Length	Informational	Resolved
MNT-32	Compatibility Issues With Mantle L2 & Solidity Version ^0.8.20	Informational	Closed
MNT-33	Proxy Inherited Solidity Contracts Should Contain A Storage Gap	Informational	Closed
MNT-34	Missing Zero Address Checks In Checkpoint Oracle	Informational	Resolved
MNT-35	TSS Node Clients Have Unreachable Max Connection Attempts Logic	Informational	Resolved
MNT-36	Irregular Size Of The First Signature In Sequence Will Trigger Panic	Informational	Closed
MNT-37	Unused Items in Structures	Informational	Resolved
MNT-38	Miscellaneous General Comments	Informational	Closed

MNT-01	Elected TSS Nodes Can Act Without Any Deposit		
Asset	`packages/contracts/contracts/L1/tss/TssGroupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A node can remove its insurance deposit and still be elected as an active TSS node.

TSS nodes are voted for by the BITDAO which then pushes the currently elected nodes on-chain. Nodes that wish to be voted for must provide a deposit as insurance that they will perform their role honestly if elected. By timing a withdrawal correctly, a node can remove their deposit and still be elected as an active TSS node. As a result, there is no means of punishing the node for inactivity or malicious behaviour.

This can occur because election results are published on-chain and so a dishonest node could watch the Ethereum mempool and frontrun the transaction including their own approval by the BITDAO with a call to `withdrawToken()`. This has the effect of removing their deposit and still being elected in the BITDAO's transaction.

As a result the node would no longer have a deposit from which to deduct funds if they are slashed; this means they are not incentivised to act honestly or timely.

Recommendations

There are several possible solutions for this issue:

- An on-chain solution could be to add a check to `setTssGroupMember()` that validates each elected node still has a sufficient deposit in the `TssStakingSlashing` contract.
- For an off-chain approach, the BITDAO could push all election results on-chain using a method such as Flashbots which avoids transactions entering the mempool where other Ethereum nodes can read them. If this approach is taken then the BITDAO must also manually check all elected nodes have not removed their deposit prior to them pushing the result on-chain.

Resolution

The issue has been addressed at PR-826.

MNT-02	No Mechanism To Flag Sequencer Fraud		
Asset	`packages/contracts/contracts/da/BVM_EigenDataLayrChain.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Note, while Mantle are aware of this and have intentionally structured the code as such, it was deemed important to comment on a feature that will eventually need to be enabled in Mantle L2.

There are currently no means of challenging a bad input from a sequencer into the Eigen Data Availability layer.

The `BVM_EigenDataLayrChain` contract contains a function `proveFraud()` that is intended to be used by any challenger to flag a data storage submission by the sequencer that is invalid. However, this currently only checks for the inclusion of a dummy string and cannot be used to flag sequencer fraud.

While the sequencer is currently a centralised role controlled by Mantle, this role will be decentralised at a later date. If the fraud proving system is not implemented, it will be possible for a sequencer to submit an invalid transaction or invalid state transition data to the Eigen Data Availability nodes with no negative repercussions. As an optimistic rollup Mantle relies on the ability to flag incorrect state transitions to deliver the security of the Mantle network.

Recommendations

Fraud proofing system needs to be fully implemented and tested prior to decentralising the sequencer role on Mantle.

Resolution

The issue has been acknowledged by the development team with the comment:

> This is an EigenDA feature. The onchain proof of custody (DA fraud proof) won't be enabled upon launch. We have the fraud_proof_string as a placeholder for later when we enable it. Without proof of custody, we use dl-retriever to detect if data is effective, and dl-dispeser to detect if a DA node is live. If da-retriever detects data is ineffective, the record will be logged on L1 smart contract recordSlash, and a new data batch will be submitted to replace the previous data. A DA node will be removed from the network if it repeats to fail the detects.

MNT-03	Default Values & Out-of-Bounds Panics In Merkle Tree Implementation		
Asset	`tss/common/merkle_tree.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The Merkle Tree implementation encounters an out-of-bounds error when processing sets with a large number of elements exceeding 131,072.

Additionally, the utilization of default hashes to fill odd-sized trees allows for the default values to then be used to successfully pass verification.

From `tss/common/merkle_tree.go` line [72]:

```
if rowSizeIsOdd {
  leftSibling = elements[rowSize-1]
  bz, _ := hexutil.Decode(defaults[depth]) // @audit, this should be assigned 0x00 instead of a default value, otherwise default
      ↪  values will pass as valid. Also it will panic with out-of-bounds error for large arrays as `len(defaults) == 16`.

  copy(buf[0:32], leftSibling[:])
  copy(buf[32:64], bz)
  elements[halfRowSize] = crypto.Keccak256Hash(buf)
}
```

Recommendations

Instead of assigning default values to padded leafs, assign `0x00` value.

This would also eliminate out-of-bounds panics, which root cause is an insufficient number of default values to use.

Resolution

The issue has been addressed at PR-1138.

Development team has also advised:

> We use defaults in tss/common/merkle_tree.go to be consistent with Lib_MerkleTree.sol, so that the same stateroots can produce the same Merkle root.

MNT-04	Elected TSS Nodes Can Avoid Slashing By Having Insufficient Deposits		
Asset	`packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

TSS nodes can avoid being slashed by manipulating their deposit reserves.

TSS nodes must provide a deposit as insurance that they will perform their role honestly if elected. It is possible for a TSS node to be elected with a deposit smaller than the slashing quantity, resulting in an inability to punish the node's misbehaviour.

This can occur either as result of a past slashing of the node or if the BITDAO chooses to change deposit minimums.

The only check on a node's deposit size is made when they initially deposit funds by calling `staking()` and so any reduction that happens thereafter does not impact the node's validity.

Therefore, when a node's deposit is decreased by slashing, no checks occur to prevent the node being elected again. Likewise, if the BITDAO chooses to raise the slashing amounts, no checks are performed on existing staked node deposits.

If such node is then elected to the TSS group, any attempt to slash them due to misbehaviour will revert due to the check on line [**287**] in `TssStakingSlashing.sol`.

Recommendations

This can be solved via an off-chain approach or by alternations made on-chain:

- The BITDAO could manually verify all TSS nodes being voted on have a sufficient deposit, blacklisting them from nomination if they do not.
- An on-chain solution could be to add a check to `setTssGroupMember()` that validates each elected node still has a sufficient deposit in the `TssStakingSlashing` contract. This would trigger a revert if one or more nodes had an insufficient deposit and this would be grounds for a new group of nodes to be elected.

Resolution

The issue has been addressed by implementing an on-chain solution at PR-826.

MNT-05	TSS Nodes Set Includes Slashed Node By Default		
Asset	`tss/manager/agreement.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Slashed node can receive portion of the distributed slashing reward.

When a TSS node is slashed some of its deposit is removed and handed to the other, honest, TSS nodes. Due to an oversight the slashed node receives some of its penalty fees back, reducing the reward that other nodes receive for being honest.

This occurs due to an oversight in the default Golang node behaviour. Currently, the TSS Manager is responsible for coordinating the TSS nodes signing and supplies the full TSS active node set to TSS nodes as the set of signing nodes. This means the node being slashed is included in the set of nodes who are then rewarded from the slashing.

While it is possible for a TSS Node to generate their own set of nodes to include in the slashing message it is unlikely a node will deviate from the default message generation behaviour as all participating signing nodes must sign the same message in order for the message to be considered valid.

Recommendations

Modify the TSS manager to remove the slashed node from the set of nodes broadcast to TSS nodes in the slashing message. Alternatively the removal of the slashed node could be included in the TSS node prior to signing the message.

The slashed node could also be removed on-chain if it is found in the set of nodes that were intended to be rewarded from the slashing. This method is more reliable however it would increase the gas costs of slashing a node.

Resolution

The issue has been closed as false-positive with the comment from development team:

> Before selecting the TSS nodes, we will query the latest active node information off-chain, and slashed nodes will be excluded. When submitting the public keys of the selected node to TssGroupManager, the TssGroupManager will double-check the deposit amounts and ensure all deposit amounts are no less than the minimum deposit amount.

MNT-06	Precompiled Contract Not Updated		
Asset	`l2geth/contracts/tssreward/contract/tssreward.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The precompiled contract `tssreward.go` in L2geth, scheduled for implementation at chain genesis, exhibits inconsistencies with its corresponding source code `packages/contracts/contracts/L2/predeploys/TssRewardContract.sol` in the parameters of certain functions and the absence of some functions, as indicated by the ABI on line [**33**].

Within the L2geth framework, a set of precompiled contracts are intended to be deployed on-chain during the chain genesis process. One of these contracts, namely `tssreward.go`, underwent its latest modification on November 11, 2022. Conversely, the corresponding source code file, located at `packages/contracts/contracts/L2/predeploys/TssRewardContract.sol` was last edited on November 23, 2022.

Detecting the precise discrepancies between the two versions poses a challenge since `tssreward.go` solely contains bytecode representation. However, a comparison of the ABI on line [**33**], reveals variations in the function parameters and the absence of certain functions in `tssreward.go`.

Updating the TSS reward contract or its address would require a hard fork as both are stored within L2geth and so would alter consensus. This hard fork could cause a chain split or further issues on-chain for contracts and users.

Recommendations

Update the predeployed Golang contract `tssreward.go` with the correct bytecode and ABI information for the newer Solidity contract.

Resolution

The issue has been acknowledged by the development team with the following comment:

> The tssreward.go has been abandoned, so the relevant version has not been updated and will be removed at a later date.

MNT-07	L2Geth Client Private Key Stored Without Encryption		
Asset	`l2geth/crypto/crypto.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The L2geth node's private key is stored in cleartext.

The functions `SaveECDSA()` and `LoadECDSA()`, which save a private key to file and load a private key from a file respectively make no use of encryption or decryption.

This suggests that the l2geth client's private key is stored in cleartext. As L2geth is used by Rollup verifiers and other system actors this is not advisable.

Recommendations

Do not store sensitive information, such as private keys, in cleartext. Implement encryption for sensitive data at rest.

Resolution

The development team has confirmed that the user's private key is stored encrypted with password as per passphrase.go#L186 and that cleartext storage was only intended for development and test environments.

MNT-08	Lack Of Signature Size Verification Checks		
Asset	`tss/ws/server/handler.go`, `tss/manager/sign.go`, `datalayr-mantle/common/contracts/utils.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Multiple instances of lack of signature's size verification before use have been identified:

- `tss/ws/server/handler.go` does not have checks to verify if signature `sigBytes` is of sufficient size:

```
178  sigBytes, sigErr := hex.DecodeString(sig)
     if pubErr != nil || sigErr != nil {
180    wm.logger.Error("hex decode error for pubkey or sig", "err", err)
       return
182  }
     digestBz := crypto.Keccak256Hash([]byte(timeStr)).Bytes()
184  if !crypto.VerifySignature(pubKeyBytes, digestBz, sigBytes[:64]) {
       wm.logger.Error("illegal signature", "publicKey", pubKey, "time", timeStr, "signature", sig)
186    return
     }
```

  If the signature is not of sufficient length line [**184**] could result in an unhandled out-of-bounds panic.

- `tss/manager/sign.go` also does not implement size checks before slicing the signature array:

```
92   if !crypto.VerifySignature(poolPubKeyBz, digestBz, signResponse.Signature[:64]) {
       log.Error("illegal signature")
94     return
     }
```

  If `signResponse.Signature` length is less than `64`, then this slicing operation `signResponse.Signature[:64]` will panic with out-of-bounds error.

- Function `CompactSignature()` in `datalayr-mantle/common/contracts/utils.go` also does not implement signature size checks on `sig` parameter, however, this particular function does not appear to be used anywhere in the codebase:

```
160  func CompactSignature(sig []byte) [64]byte {
       // Decompose sig
162    r := Copy32Byte(sig[:32])
       s := Copy32Byte(sig[32:64])
164    v := sig[64]
```

Recommendations

Implement additional checks on the signature length prior to taking a slice of the signature bytes or performing any indexing operations.

Resolution

The issue has been addressed at PR-1131.

MNT-09	Incorrect Error Returned With Empty Signature		
Asset	`tss/node/tsslib/keysign/tss_keysign.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`generateSignature()` may return an empty signature data in a successful response.

In `SignMessage()` of `tss/node/tsslib/keysign/tss_keysign.go`, if an error is detected when calling `SetupIDMaps()` an empty signature data field is returned. However, it is returned with an incorrect error object (`err` instead of `err1` or `err2`), meaning that an empty signature will be erroneously used later on:

```
99  err1 := conversion.SetupIDMaps(partyIDMap, tKeySign.tssCommonStruct.PartyIDtoP2PID)
    err2 := conversion.SetupIDMaps(partyIDMap, abnormalMgr.PartyIDtoP2PID)
101 if err1 != nil || err2 != nil {
      tKeySign.logger.Error().Err(err).Msgf("error in creating mapping between partyID and P2P ID")
103   return emptySignatureData, err  // @audit this should be either err1 or err2
    }
```

Then the check in `tss/node/tsslib/keysign.go:generateSignature()` line [**48**] will fail (due to `err` being `nil`), returning an empty signature in the successful response on line [**59**]:

```
44    signatureData, err := keysignInstance.SignMessage(req.Message, localStateItem, signers)
      // the statistic of keygen only care about Tss it self, even if the following http response aborts,
46  // it still counted as a successful keygen as the Tss model runs successfully.

48  if err != nil {
      t.logger.Error().Err(err).Msg("err in keysign")
50    culprits := keysignInstance.GetTssCommonStruct().GetAbnormalMgr().TssCulpritsNodes()
      return keysign2.Response{
52      Status:     common.Fail,
        FailReason: abnormal.SignatureError,
54      Culprits:   culprits,
      }, nil
56  }

58  return keysign2.NewResponse(
      &signatureData,
60    common.Success,
      "",
62    nil,
      ), nil
```

Recommendations

Correct `tss/node/tsslib/keysign/tss_keysign.go` line [**103**] to return the relevant `err1` or `err2` instead of `err` .

Resolution

The issue has been addressed at PR-1131.

MNT-10	Lack Of Array Size Checks Before Slicing Or Referencing Array Elements		
Asset	`mt-challenger/*`, `datalayr-mantle/*`, `fraud-proof/*`, `batch-submitter/*`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There are instances of dynamically referencing array's elements, or slicing arrays, without prior checks of the array size.

This could trigger unhandled index out-of-bounds panics, crashing the program.

- `mt-challenger/challenger/challenger.go`:

```
272 polys, multirevealProofs, batchPolyEquivalenceProof, err :=
    dp.ProveBatchInterpolatingPolyDisclosure(frames[startingChunkIndex:endingChunkIndex+1],
274 store.DataCommitment, store.Header, uint32(startingChunkIndex))
```

  There are no checks to ensure `startingChunkIndex` and `endingChunkIndex+1` are within `frames[]` size.

- `datalayr-mantle/common/contracts/challenges.go`:

```
198 interpolationPoly := frame.Coeffs
    numNode := ceilNextPowOf2(ceilNextPowOf2(header.NumSys) + header.NumPar)
200 interpolationPolyCommit := bn254.LinCombG1(ks.SecretG1[:len(interpolationPoly)], interpolationPoly)
```

  This may panic if `len(interpolationPoly)` is greater than `len(ks.SecretG1)`.

- `datalayr-mantle/common/contracts/utils.go`:

```
89  func BigIntToBytes(r *big.Int, num int) []byte {
      b := r.Bytes()
91
      t := make([]byte, num)
93
      for i := 0; i < len(b); i++ {
95      t[num-1-i] = b[len(b)-1-i]
      }
97
      return t[:]
99  }
```

  No checks to ensure `num >= len(b)`, otherwise assignment on line [**95**] will panic due to `i` counter eventually being larger than `num` and resulting with negative value when referencing `t[num-1-i]`.

- `datalayr-mantle/lib/kzgFFT/kzgRs/decode.go` in functions `Decode()` and `DecodeSys()`:

```
89 return concatFr[:inputSize], nil
```

  There are no checks to ensure `inputSize` is within size limit of `concatFr`.

- `datalayr-mantle/lib/merkzg/merkzg.go`:

```
78 func (mt *KzgMerkleTree) ProveIndex(index int) [][]byte {
   proof := make([][]byte, 0)
80 height := len(mt.Tree) - 1
   tmp := index
82 for height > 0 {
     if tmp%2 == 0 {
84     proof = append(proof, mt.Tree[height][tmp+1])
     } else {
86     proof = append(proof, mt.Tree[height][tmp-1])
     }
88   tmp /= 2
     height--
90 }
   return proof
92 }
```

No checks to ensure that `index` is within tree's bounds on line [**84**] and line [**86**].

- `datalayr-mantle/dl-node/validation.go`:

```
75   func (s *Server) verifyLowDegreeProof(polyCommit, shiftedCommit *bn254.G1Affine, shift int) bool {

77   polyG1 := (*wbls.G1Point)(polyCommit)
     shiftedG1 := (*wbls.G1Point)(shiftedCommit)
79
     s.logger.Debug().Msgf("LOW DEGREE VER", &s.KzgVerifier.S2[len(s.KzgVerifier.S2)-shift], len(s.KzgVerifier.S2)-shift)
81
     return wbls.PairingsVerify(polyG1, &s.KzgVerifier.S2[len(s.KzgVerifier.S2)-shift], shiftedG1, &wbls.GenG2)
83 }
```

No checks to ensure that `len(s.KzgVerifier.S2)-shift` is larger than or equal to `0`.

- `fraud-proof/rollup/types/tx_batch.go`:

```
49 func (b *TxBatch) LastBlockNumber() uint64 {
     return b.Contexts[len(b.Contexts)-1].BlockNumber
51 }

53 func (b *TxBatch) LastBlockRoot() common.Hash {
     return b.Blocks[len(b.Blocks)-1].Root()
55 }
```

If `b.Contexts` or `b.Blocks` size is 0, this will panic.

- `fraud-proof/proof/state/state.go`:

```
247 transactionTrie := NewTransactionTrie(transactions[:transactionIdx])
    receiptTrie := NewReceiptTrie(receipts[:transactionIdx])
```

If `transactionIdx` is larger than `len(transactions)`, this will panic.

Note, all places that this function was called from in the reviewed codebase passed in `transactionIdx` within a valid range.

- `fraud-proof/proof/state/stack.go`:

```
71 func (st *Stack) Hash() common.Hash {
     return st.hash[len(st.hash)-1]
73 }

75 func (st *Stack) PopN(n int) {
     st.data = st.data[:len(st.data)-n]
77   st.hash = st.hash[:len(st.hash)-n]
   }
```

```
88  func (st *Stack) Back(n int) *uint256.Int {
      return &st.data[st.Len()-n-1]
90  }

92  func (st *Stack) HashAfterPops(n int) common.Hash {
      return st.hash[st.Len()-n]
94  }
```

No checks implemented to verify if `n` is within `len(st.data)` or `len(st.hash)`, or if `len(st.data)` and `len(st.hash)` have at least 1 element in it.

- `fraud-proof/proof/proof/instructions.go`:

```
1086  createdCode := currState.Memory.Data()[offset : offset+size]
```

If `offset` or `offset+size` is larger than size of `Data` array, this will panic.

- `batch-submitter/drivers/sequencer/encoding.go`:

```
func readUint64(r io.Reader, val *uint64, n uint) error {
  var buf [8]byte
  if _, err := r.Read(buf[8-n:]); err != nil {   // @audit 'n' needs to less than 8, otherwise could trigger panic.
    return err
  }
  *val = byteOrder.Uint64(buf[:])
  return nil
}
```

There are no checks that the final `n` parameter of `ReadUint64()` is less than or equal to 8.

Note, this instance did not appear to be exploitable in the reviewed codebase as everywhere it was called from, the value of passed in `n` was less than 8.

Recommendations

Review all instances in the codebase and, where feasible, implement additional array size checks before slicing or dynamically referencing array's elements.

Resolution

The issue has been addressed at PR-1141.

Additionally, the development team advised the following:

> Datalayr-mantle/*: Deprecated.
>
> fraud-proof/*: Reject. The depth of EVM stack will be greater than or equal to that of popNum. When a transaction is executed on Layer 2, EVM execution of fraud proof generation will be consistent with L2geth. Therefore it doesn't require a length check. It is the same as EVM implementation on Ethereum.
>
> Batch-submitter/*: Add range check.

MNT-11	Lack Of Size Checks In `createUUID()` Function		
Asset	`tss/node/tsslib/storage/shamir_mgr.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Generation of UUID from a public key may result in unhandled out-of-bounds panic.

There are no size checks implemented to ensure `len(keyBytes) > 16`:

```
548     func createUUID(key string) (string, error) {
        uuidBytes := make([]byte, 16)
550     keyBytes := []byte(key)
        copy(uuidBytes, keyBytes[len(keyBytes)-16:len(keyBytes)])
```

If `len(keyBytes)` is less than 16, the `keyBytes[len(keyBytes)-16:len(keyBytes)]` will trigger an unhandled panic.

As the public key is taken from `keysign2.Request` of external source via TSS node's `keysignHandler()`, this could potentially be triggered via a malformed incoming request.

Recommendations

Implement checks to verify that `len(keyBytes) > 16` prior to taking a slice of `keyBytes`.

Resolution

The issue has been addressed at PR-1131.

MNT-12	Lack Of Size Validation Of Byte Arrays Used With BLS		
Asset	`datalayr-mantle/common/crypto/bls/bls.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There are no checks implemented in `DeserializeG1()` and `DeserializeG2()` functions of `datalayr-mantle/common/crypto/bls/b` to verify if provided byte array is of sufficient size:

```
205 func DeserializeG1(b []byte) *bn254.G1Affine {
    p := new(bn254.G1Affine)
207 p.X.SetBytes(b[0:32])
    p.Y.SetBytes(b[32:64])
209 return p
}
```

```
233 func DeserializeG2(b []byte) *bn254.G2Affine {
    p := new(bn254.G2Affine)
235 p.X.A0.SetBytes(b[0:32])
    p.X.A1.SetBytes(b[32:64])
237 p.Y.A0.SetBytes(b[64:96])
    p.Y.A1.SetBytes(b[96:128])
239 return p
}
```

If `b` parameter array's size is smaller than 64 for `DeserializeG1()` or 128 for `DeserializeG2()`, the function will panic with index out-of-bounds error.

This could be manipulated by dispersers by using `StoreFramesRequest` via `StoreFrames()` in `datalayr-mantle/dl-node/server.go` line [**174**].

Recommendations

Implement an additional check to verify that size of the array passed in as `b` parameter is larger than an expected minimum.

Resolution

The development team has advised:

> Please note that the prior version of the EigenDa code has been deprecated. The issue does not apply to the new version.

MNT-13	TSS Nodes Reporting Slashing Are Vulnerable To Front Running		
Asset	`packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Reporting nodes can be front run and miss out on rewards.

TSS nodes are responsible for broadcasting off-chain node slashing decisions on-chain, however it is possible for any other user to read this transaction information while it is in the Ethereum mempool and frontrun the report. This will result in the reporting node not receiving the reward they are entitled to for sending the transaction.

This happens because the information needed to report a node slashing does not contain any information unique to the sender, meaning it can be copied and if the new sender submits a higher gas price their transaction will be included prior to the original node sender. Then, the main slashing reward is allocated to `msg.sender` on line [**306**] in `packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`.

This conflicts with Mantle's intention which is outlined in documentation as *"rewards the tss-node that submits the slashing message, and rewards the person who participates in the report"*.

Recommendations

One possible solution would be to make TSS node operators aware that this can happen and that they need to use a submission mechanism that does not display their transaction to the Ethereum mempool such as flashbots.

Alternatively, the node who reported the slashing off-chain should be stored as part of the signed message, this way then it can be used to verify the addressing being rewarded matches the reporting node. While this could cause problems if this node then goes offline, and so is unable to report on-chain, other participating nodes do have an incentive to report the issue as they receive a (smaller) bounty too.

Resolution

The issue has been resolved in PR #826. The resolution is to pay all slashing rewards directly to a regulatory account.

On line [**309**] the second parameter to `slashShares()` has been modified to `regulatoryAccount` which represents the address to receive payouts from a slashing.

```
TssDelegationManager(tssDelegationManagerContract).slashShares(stakerS[i], regulatoryAccount, delegationShares,tokens,
    ↪  delegationShareIndexes, shareAmounts);
```

MNT-14	TSS Manager Is A Single Point Of Failure		
Asset	`tss/manager/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The TSS manager represents a centralised point of failure within the system as it is responsible for coordinating and facilitating communication among TSS nodes.

The threshold signature scheme (TSS) relies on a TSS manager role to coordinate TSS nodes for signing messages to verify transaction batches produced by the sequencer and slash inactive or malicious TSS nodes. Without the TSS manager, state roots cannot be transitioned from Mantle L2 to be recorded on Ethereum L1.

Furthermore, TSS nodes have no coordinator to dictate which slashing messages to sign, as these must match between TSS nodes prior to being included on-chain.

In addition to this, the TSS manager must communicate with all prospective TSS nodes, meaning it is a likely target for DoS attacks.

Recommendations

Given the importance of the TSS manager role, look at methods of decentralising this role among the TSS nodes to make it resilient.

If TSS manager is to be kept centralised, changes could be made to the TSS manager to ensure its additional resilience. One of the approaches could be having a back-up TSS manager who TSS nodes can ping if they cannot communicate with the main TSS manager. The TSS manager could also block connections from nodes who are not actively involved in the current election round as a defensive measure.

Resolution

The issue has been acknowledged by the development team with the following comment:

> The suggestion is helpful. An emergency backup plan exists for the TSS network, which will allow us to quickly recover the network in the event of stability issues. Additionally, measures are in place to punish inactive nodes. Therefore, the potential for a single-point failure of the TSS manager is not a predominant concern.

MNT-15	Sensitive Information Passed In Command Line Arguments		
Asset	`mt-batcher/cmd/mt-batcher/main.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The application accepts flags containing sensitive information, such as private keys, and mnemonics as input parameters through the command line interface.

The underlying system's history file, typically accessible by authorized users or administrators, retains a record of executed commands and their associated command line arguments. Consequently, any flags containing sensitive information, become persistently stored in cleartext, in the history file.

This poses a significant risk as unauthorized users or malicious actors with access to the system history file could extract and misuse the sensitive information.

Recommendations

Use Golang's `x/term` package (`https://pkg.go.dev/golang.org/x/term@v0.7.0#ReadPassword`) to read a line of sensitive input from a terminal, without local echo.

Resolution

The issue has been acknowledged by the development team with the following comment:

> For Mantle mainnet, sensitive information is sharded to mutiple parts, encrypted by AWS KMS and separately stored in AWS SecretsManager. We use CloudHSM for secp256k1 signing. For development and testing environments, we will store sensitive information locally offline.

MNT-16	Sequencer Address Updates Can Immobilise Contract		
Asset	`packages/contracts/contracts/da/BVM_EigenDataLayrChain.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The sequencer may update itself with an incorrect address, resulting in the loss of its ability to execute sequencer-only functions.

The `BVM_EigenDataLayrChain` contract contains a sequencer role which is responsible for updating various addresses stored within the contract. This sequencer can call a function to update itself but as this is a one-stage process it is possible to update with the wrong address and so lose the ability to call any sequencer only function.

Consequently, there is a risk where the sequencer may inadvertently update itself with an incorrect address, rendering it incapable of invoking any functions designated exclusively for sequencer use.

Recommendations

Make updating the sequencer a two-stage process where the new sequencer is proposed, and then the new sequencer accepts the role in another transaction.

Resolution

The issue has been addressed at commit 91bd160.

MNT-17	TSS Node Design Uncertainty		
Asset	`tss/slash/slash.go`, `packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Further consideration is required on the intended behaviour of slashing inactive TSS nodes.

Documentation in `"TSS Enhancement Spec.pdf"` and `TssStakingSlashing.sol` suggest that inactive TSS nodes should be slashed by their peers to promote timely responses. The TSS manager and TSS nodes both appear to run an indexer that handles flagging TSS nodes as inactive.

However, communications with the Mantle team have left uncertainty about this design, as it was stated that neither the TSS manager, nor the TSS nodes would record TSS nodes who were inactive.

If TSS nodes are not slashed for inactivity then this presents a problem with ensuring a quorum of TSS nodes is available for message signing.

On the other hand, if nodes are reported for inactivity, then this presents a different risk. TSS nodes have an incentive to obstruct each other's activity due to slashing rewards. As nodes communicate with one another during the signing process they have the ability to discover each other's locations. This means nodes are likely to perform DDoS attacks on one another, degrading the performance of the TSS system.

Recommendations

Review the design for reporting inactive TSS nodes and ensure all resulting behaviours are intended and risks mentioned above carefully considered.

Update the documentation and code comments accordingly.

Resolution

The specification document has been updated to reflect the latest implementation details.

In the latest design, `Tssmanager` initiates the slash transaction which is verified by all `Tssnodes` rather than individual `Tssnodes` packaging the transaction.

Additionally, to prevent incentives to slash nodes which are not misbehaving the receiving address of slashing rewards is a `regulatoryAccount` rather than `msg.sender`. This can be seen on line [**309**] of `packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`.

MNT-18	If TSS Nodes Fall Below Quorum Size No TSS Decisions Can Be Made		
Asset	`packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

If too many TSS nodes are removed from the signing group due to bad behaviour it may become impossible to reach a quorum with the remaining nodes. This would lead to the TSS nodes no longer being able to sign any messages, such as sequencer state roots or slashing of other TSS nodes.

Not being able to sign messages would degrade the trust in Mantle's system as the TSS nodes would no longer be able to confirm the published state roots were correct.

While it would be possible to transition to a new group public key created by a new group of TSS nodes, this process would take time as the BITDAO would need to organise a vote for a new group of TSS nodes. Then, it would need to push the results of said vote on-chain to be actioned by the nodes.

Because of Mantle's intention for the set of TSS nodes to be decentralized and egalitarian, with nominated nodes only needing to have staked BIT tokens, it is likely there will be a range of experience in node operator skills. This may manifest in node downtime or incorrect signature choices that would impact the number of nodes available to reach quorum on a vote.

Recommendations

There are multiple approaches that could mitigate this issue:

- A larger TSS node group could have a lower quorum without degrading security. For example, if Mantle has an 8 out of 10 node quorum, this group could be expanded to an 8 out of 12 quorum. While the percentage of agreeing nodes has dropped to 66% from 80% the group still requires a majority opinion to act and would require the loss of 5 nodes rather than 3 before it could no longer sign messages.
- Quorum requirements could remain the same percentage of nodes but have more nodes could be elected per round. This would increase the quantity of nodes that would need to experience failure before the voting quorum becomes unreachable.
- A set of backup nodes are voted on during the BITDAO node election. These nodes do not form part of the original group signature but in the event that the group cannot reach quorum then a new group signature can quickly be formed with the backup nodes taking the place of the nodes removed due to bad behaviour.

Resolution

The resolution occurs in PR #826. Nodes are to be immediately jailed upon slashing as seen in the following code snippet.

```
254 function slash(SlashMsg memory message) internal {
        // slashing params check
256     require(isSetParam,"have not set the slash amount");
        bytes memory jailNodePubKey = operators[message.jailNode];
258     if (message.slashType == SlashType.uptime) {
            // jail and transfer deposits
260         ITssGroupManager(tssGroupContract).memberJail(jailNodePubKey); //@audit jail
            transformDeposit(message.jailNode, 0);
262     } else if (message.slashType == SlashType.animus) {
            // remove the member and transfer deposits
264         ITssGroupManager(tssGroupContract).memberJail(jailNodePubKey); //@audit jail
            transformDeposit(message.jailNode, 1);
266     } else {
            revert("err type for slashing");
268     }

270
    }
```

Off-chain monitoring is put in place to determine when a node has been jailed and the development team have provided the following comment.

> Mantle devops team will monitor the liveness of Tssnodes, and will notify the jailed node to initiate a transaction to unjail itself when it's jailed. Mantle devops team will run all the Tssnodes upon Mantle mainnet launch, and will handle the slashing incidents promptly.

MNT-19	TSS Nodes Could Be Paid The Same Fee Twice		
Asset	`packages/contracts/contracts/L2/predeploys/TssRewardContract.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

It is possible for TSS nodes to receive pending fees for a period twice if a call to trigger `claimRewards()` on Mantle L2 is repeated on Ethereum L1.

While unlikely, in the event that `claimRewards()` is called twice while transitioning the Mantle system to burning fees, it is possible for the fees to be paid out by both pathways of fee payment. Due to the different checks that then take place both pathways will succeed and send funds to the TSS nodes when only one pathway should succeed.

The situation could occur due to a software/user error or low gas delaying the execution. This would lead to a duplicated call created on L1 Ethereum which are detected and acted upon on Mantle L2. These transactions, one of which is delayed, could then sandwich a transaction to enable fee burning on Mantle L2 by mistake.

On the first call, before fee burning is activated the fees would correctly be paid out via `claimRewardByBlock()` on line [**97**]. Then, after `BVM_GasPriceOracle.setIsBurning()` has been called on Mantle L2, the second identical call would propagate. When fee burning is enabled a different accounting system is used to determine if fees have already been paid to the TSS nodes, this means the TSS nodes would be paid twice for the same period.

Recommendations

There are several approaches that could prevent this from occurring in future:

- Update `lastBatchTime = _batchTime;` on both logic branches, adding this prior to the return on line [**106**] will ensure the second call reverts.
- Careful control of input variables would prevent this situation occurring. If `_batchTime` is only set to be non-zero when the system is not burning fees then the second call would fail due to a check on line [**108**].

Resolution

The issue is resolved by first removing the functions `claimRewardByBlock()` and `BVM_GasPriceOracle.setIsBurning()` as seen in PR #826. Additionally, the development team have provided the following comments describing how `claimReward()` will be call through the cross-chain messages, preventing re-org attacks.

> We call claimReward on Mantle Network (L2) by cross-chain messages from L1. We wait for 64 blocks on L1 to avoid replaying transactions due to L1 reorg. In case of L1 reorg, batchAmount = (_batchTime - lastBatchTime) * querySendAmountPerSecond() will return 0.

MNT-20	TSS Nodes Can Report Themselves For Slashing To Reduce Penalty		
Asset	`packages/contracts/contracts/L1/tss/TssStakingSlashing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

If a TSS node has performed behaviour that is deserving of slashing they can lessen the impact of this by self reporting. This will reduce the penalty they pay for inactivity or malicious behaviour as they would then receive the slashing reward for having issued the report.

Self-slashing is possible because in `transformDeposit()`, which is called by `slashing()`, there is no check on who the reporting `msg.sender` is. Therefore, it is possible that the person being slashed could be the transaction sender.

Mantle documentation highlights this should not be possible *"Slashing: Any tss-node reports tss-nodes other than him[self]"*. By reducing the overall slashing fee a TSS node must pay there is less economic incentive for them to perform their elected role and this could have detrimental effects on the stability of the Mantle network as the guarantees of the TSS node role are less likely to be upheld.

Recommendations

Add a check to `transformDeposit()` such as `require(msg.sender != deduction, "Sender cannot be slashed node")`. However, without the fixes noted in MNT-13 the slashed node could also propose from another unrelated account so it is advised to solve MNT-13 also.

Alternatively, code could be added to the TSS manager to prevent a node from nominating itself for slashing, this solution would also have to be coupled with the solution to MNT-13 to ensure the slashed node does not frontrun and copy a transaction electing to slash it.

Resolution

Updated functionality has been implemented in PR #826 and the development team have provided the following comment:

> Significant modifications have been made to TSS rewards and slashing mechanism. In the latest design, Tssmanager initiates the slash transaction and it gets verified by all Tssnodes. Tssmanager will transfer the penalty to a dedicated penalty receiving address, instead of sending the penalty as reward to the reporting Tssnode.

By changing the receiver address to `regulatoryAccount` in `transformDeposit()` on line [**309**], the `msg.sender` will no longer receive the slashing reward. Hence, nodes are not motivated to report themselves for slashing.

MNT-21	Hardcoded Addresses Of `BIT` Token		
Asset	`Various files`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Several instances of hardcoded `BIT` token addresses were identified in the code.

BITDAO has decided to transition from the `BIT` token to `MNT` as per the guidelines outlined in BIP-21. This transition is expected to take place before the launch of Mantle L2. However, there are instances within the contracts where addresses are hardcoded to point to the `BIT` token, which can lead to compatibility issues and unexpected behavior during the token transition.

Identified contracts with hardcoded addresses include (note, the list is non-exhaustive):

- line [**199**] in `L1StandardBridge`
- line [**21**] in `BVM_BIT`

Recommendations

Correct all instances of hardcoded `BIT` addresses in the codebase.

Resolution

The issue has been addressed at PR-1075.

Additionally, the development team has advised the following:

> We will provide a token migration contract to help users to migrate their BIT token to MNT token on Ethereum L1.
>
> Some of the contracts mentioned in the feedback use old BIT tokens, but our bridge-related contracts use new MNT tokens, causing incompatibility issues. According to our plans, our bridge will not support the BIT bridge function, only MNT. Before using our bridge for these BITs stored in contracts, users must withdraw their BIT into their EOA accounts and complete the migration. If these contracts can't update their token address to MNT token, then these contracts should be abandoned.

MNT-22	Lack Of Upper Bounds Checks On `fraudProofPeriod`		
Asset	`packages/contracts/contracts/da/BVM_EigenDataLayrChain.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The absence of upper bounds on the `fraudProofPeriod` parameter introduces a risk of overflow and may result in an instantly confirmable proof.

The addition operation on line [**300**] `uint32(block.timestamp + fraudProofPeriod)` is performed as `uint256` and then cast to `uint32`, resulting in potential overflow for values greater than or equal to `2^32`.

Although setting an extremely large fraud proof period is unlikely, it is important to address this issue to ensure system integrity.

Recommendations

Implement upper bounds or validation checks on the `fraudProofPeriod` parameter to ensure it does not exceed the maximum value that can be represented by a `uint32`.

Perform the addition operation with appropriate data types that can handle larger values, such as `uint256`, without the risk of overflow.

Resolution

The issue has been closed with the reasoning that the longest time to prove fraud is seven days and it is unlikely that `uint32(block.timestamp + fraudProofPeriod` will exceed `type(uint256).max` in foreseeable future.

MNT-23	Gas Exhaustion In `resetRollupBatchData()` Loop		
Asset	`packages/contracts/contracts/da/BVM_EigenDataLayrChain.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `resetRollupBatchData()` lacks bounds on the for loop on line [**198**], which may lead to potential gas exhaustion.

Without defining limits, the loop may iterate over many elements, causing excessive gas consumption and potential out-of-gas errors.

Recommendations

Determine a reasonable upper bound or limit for the loop based on the expected maximum value of `rollupBatchIndex`. This upper bound should ensure that the loop does not consume excessive gas or cause out-of-gas errors.

Update the loop condition to enforce the defined upper bound or limit, ensuring that the loop terminates within a reasonable number of iterations.

Resolution

The issue has been acknowledged by the development team with the following comment:

> We can specify the starting _rollupBatchIndex that needs to be reset, so that it can be reset from the latest rollupBatchIndex to the current specified _rollupBatchIndex. When many batches are required to be reset, it can be reset in batches, so that there is no out of gas error.

An event has been added on completion of `resetRollupBatchData()` at PR-1174.

MNT-24	Unreachable Error Handling In Proxyd Cache		
Asset	`proxyd/cache.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Cache misses may never be counted, leading to inaccurate logging.

There is no way to reach the `RecordCacheMiss()` function on line [**151**] as can be seen by the conflicting nested if statements.

```
149 if res != nil {
      if res == nil {
151     RecordCacheMiss(req.Method)
      } else {
153     RecordCacheHit(req.Method)
      }
155 }
```

This will lead to only `RecordCacheHit()` ever being processed and cache misses would never be counted.

Recommendations

Remove the outer `if res != nil` check on line [**149**].

Resolution

The issue has been addressed at PR-1137.

MNT-25	Mnemonic Cannot Be Used Without Supplying Private Key		
Asset	`mt-batcher/mt_batcher.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Mnemonic cannot be used without private key, defeating the purpose of supporting mnemonic phrases.

`cfg.PrivateKey = ""` is required when a mnemonic phrase is set using `GetConfiguredPrivateKey()` in `mt-batcher/services/common/crypto.go`, however, `cfg.PrivateKey` also needs to be a valid private key in `mt-batcher/services/mt_batcher.go`):

```
94  mtBatherPrivateKey, err := crypto.HexToECDSA(strings.TrimPrefix(cfg.PrivateKey, "0x"))
    if err != nil {
96    return nil, err
    }
```

```
106 feePrivateKey, err := crypto.HexToECDSA(strings.TrimPrefix(cfg.FeePrivateKey, "0x"))
    if err != nil {
108   return nil, err
    }
```

As such, it is impossible to use mnemonic without supplying private key.

Recommendations

Modify implementation to allow for use of either mnemonic or private key with `mtBatherPrivateKey` and `feePrivateKey` variables.

Alternatively if support for mnemonic is not required, remove the unused code.

Resolution

The issue has been addressed at commit bb394d0.

MNT-26	Nil Pointer Panic If DTL Returns Malformed Data		
Asset	`mt-batcher/services/client/client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

An unhandled panic may occur if `enqueue.Origin` is `nil` since it is a pointer type.

```
38 func (c *Client) GetEnqueueByIndex(index uint64) (string, error) {
   str := strconv.FormatUint(index, 10)
40 response, err := c.client.R().
     SetPathParams(map[string]string{
42     "index": str,
     }).
44   SetResult(&rollup.Enqueue{}).
     Get("/enqueue/index/{index}")
46 if err != nil {
     return "", fmt.Errorf("cannot fetch enqueue: %w", err)
48 }
   enqueue, ok := response.Result().(*rollup.Enqueue)
50 if !ok {
     return "", fmt.Errorf("cannot fetch enqueue %d", index)
52 }
   if enqueue == nil {
54   return "", fmt.Errorf("cannot deserialize enqueue %d", index)
   }
56 return enqueue.Origin.String(), nil    // @audit the function here is 'func (a Address) String() string' which must dereference
       ↪  'a' from a pointer so the nil pointer exception arises.
}
```

Recommendations

Ensure `enqueue.Origin` is not `nil` and return an error otherwise.

Resolution

The issue has been addressed at PR-1163.

MNT-27	Unhandled Panic If Metadata Is Set To Nil		
Asset	`mt-batcher/services/restorer/handle.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Decoding a pointer with `json.Unmarshal()` may result in panic if metadata is set to `nil`:

```
150     txDecodeMetaData := new(eigenda.TransactionMeta)
        err = json.Unmarshal(newBatchTxn[i].TxMeta, txDecodeMetaData)  // @audit may panic on nil txDecodeMetaData
152     if err != nil {
          log.Error("Unmarshal json fail")
154     }
```

Recommendations

Ensure `txDecodeMetaData` is not `nil` before passing it to `json.Unmarshal`.

Resolution

The issue has been addressed at PR-1142.

MNT-28	Lack of Error Handling in `Hex2Bytes()`		
Asset	`l2geth/common/bytes.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Lack of error checks in `common.HexToAddress()` may result in an undefined return value for malformed input as no errors are handled or propagated from downstream `Hex2Bytes()`:

```
95 func Hex2Bytes(str string) []byte {
     h, _ := hex.DecodeString(str)
97   return h
   }
```

Recommendations

Catch and propagate an error if one arises from `hex.DecodeString()` in `Hex2Bytes()`, e.g.:

```
h, err := hex.DecodeString(str)
if err != nil {
  return nil, err
}
return h, nil
```

Resolution

The issue has been acknowledged by the development team with the following comment:

> We have reviewed the code related to this function. It has no security implication if this err is not handled and it returns nil. Therefore we will leave it as pending until future releases.

MNT-29	Incorrect Usage Of Context In `NewDriver()` & `NewDataService()`		
Asset	`mt-batcher/services/sequencer/driver.go`, `mt-batcher/services/restorer/driver.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The context is made `WithTimeout()` and then cancelled via `defer`, which will not terminate the original context, but its sub-context.

Calling `context.WithTimeout()` will return a new sub-context and a cancel function. The sub-context will expire in `common4.DefaultTimeout` amount of time. Calling `cancel()` will end the sub-context, as opposed to the original `ctx`.

```
110     _, cancel := context.WithTimeout(ctx, common4.DefaultTimeout)
    defer cancel()
```

```
137     return &Driver{
        Cfg:           cfg,
139     Ctx:           ctx,
        WalletAddr:    walletAddr,
141     FeeWalletAddr: feeWalletAddr,
        GraphClient:   graphClient,
143     DtlClient:     dtlClient,
        txMgr:         txMgr,
145     LevelDBStore:  levelDBStore,
        FeeCh:         make(chan *FeePipline),
147     cancel:        cancel,
    }, nil
```

Recommendations

Create a sub context with a cancel then pass that to the driver, e.g.

```
subCtx, cancel := context.WithCancel(ctx)
// @audit do not `defer cancel()` as it's called in `Stop()`

// ... snipped ...

return &Driver{
  Cfg:           cfg,
  Ctx:           subCtx, // @audit patch this to use sub context
  WalletAddr:    walletAddr,
  FeeWalletAddr: feeWalletAddr,
  GraphClient:   graphClient,
  GraphqlClient: graphqlClient,
  DtlClient:     dtlClient,
  txMgr:         txMgr,
  LevelDBStore:  levelDBStore,
  FeeCh:         make(chan *FeePipline),
  cancel:        cancel,
}, nil
```

Resolution

The issue has been acknowledged by the development team with the following comment:

> Thank you for your recommendation. Given its non-critical nature, we will include it as a pending task for future releases.

MNT-30	Waitgroup Counter Not Incremented		
Asset	`mt-batcher/services/sequencer/driver.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Wait group calls `Done()` in `CheckConfirmedWorker()` and `RollupFeeWorker()` functions, but does not increment the wait group counter:

```
646 func (d *Driver) Start() error {
      d.wg.Add(1)
648   go d.RollupMainWorker()
      if d.Cfg.CheckerEnable {
650     batchIndex, ok := d.LevelDBStore.GetReRollupBatchIndex()
        log.Info("get latest batch index", "batchIndex", batchIndex, "ok", ok)
652     if batchIndex == 0 || !ok {
          d.LevelDBStore.SetReRollupBatchIndex(1)
654     }
        go d.CheckConfirmedWorker() //@audit should do d.wg.Add(1)
656   }
      if d.Cfg.FeeModelEnable {
658     go d.RollUpFeeWorker() //@audit should do d.wg.Add(1)
      }
660   return nil
    }
```

```
751 // ... snipped
    func (d *Driver) CheckConfirmedWorker() {
753   defer d.wg.Done()
        // ..
```

```
720 //... snipped
    func (d *Driver) RollUpFeeWorker() {
722   defer d.wg.Done()
        // ...
```

Recommendations

Call `d.wg.Add(1)` before calling `RollupFreeWoker()` or `CheckConfirmedWorker()`.

Resolution

The issue has been addressed at commit bb394d0.

MNT-31	`hashAfterPops()` Uses Incorrect Length
Asset	`fraud-proof/proof/state/stack.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `HashAfterPops()` function uses incorrect length when referencing elements of `hash` array.

The function code calls `st.Len()`, which references `len(st.data)` rather than `len(st.hash)`:

```
92 func (st *Stack) HashAfterPops(n int) common.Hash {
     return st.hash[st.Len()-n]
94 }
```

Note, this issue did not appear to be exploitable in the reviewed codebase as the sizes of `data` and `hash` arrays should always be equal.

Recommendations

Change the function to reference size of `hash` array instead, e.g.:

```
st.hash[len(st.hash)-n]
```

Resolution

The issue has been addressed at PR-1133.

MNT-32	Compatibility Issues With Mantle L2 & Solidity Version ^0.8.20
Asset	`Various files`
Status	**Closed:** See Resolution
Rating	Informational

Description

Contracts deployed on Mantle face compatibility issues when using a Solidity version of 0.8.20 or greater due to the absence of the OP Code `PUSH0`.

Specifically, contracts listed within the `packages/contracts/contracts/L2` directory, which have a floating pragma that may include 0.8.20, must exercise caution to ensure their functionality is not compromised.

Recommendations

Amend instances of pragma versioning on files intended to be deployed to Mantle L2 to exclude versions greater or equal to 0.8.20.

Additionally, it is recommended to add similar advice to Mantle developer documents for developers looking to build projects on Mantle.

Resolution

The issue has been acknowledged by the development team with the following comment:

> Given the novelty of the current geth version, there is no support for the opcode PUSH0. Mantle network employs pre-compiled contracts from solc 0.8.9. The primary risk resides in executing user's dAapp contracts. Consequently, we will preliminarily highlight this in our documentation, with subsequent updates introducing pertinent restrictions.

MNT-33	Proxy Inherited Solidity Contracts Should Contain A Storage Gap
Asset	`Various files`
Status	**Closed:** See Resolution
Rating	Informational

Description

If a contract is inherited by another contract that is intended to live behind a proxy, it is advisable to include a storage gap. This is an unused state variable array that provides space between the used state variables of the base contract and the storage slots used by state variables in the derived contract.

This will prevent the reorganising of storage slots in the derived contract should an additional state variable be added to the base contract. When working with proxied contracts this reorganisation can be dangerous as it will lead to the contract's logic pointing to the wrong state variables which can lead to unexpected behaviour.

Affected files include:

- packages/contracts/contracts/libraries/bridge/CrossDomainEnabled.sol
- packages/contracts/contracts/libraries/eigenda/Parse.sol
- packages/contracts/contracts/libraries/resolver/Lib_AddressManager.sol (This contract is itself fine but imports OpenZeppelin's `Ownable` when it should use `OwnableUpgradable` instead.)

Recommendations

Include an unused state variable array in this contract such as `uint256[50] __gap;`, if additional state variables are needed, reduce the size of the unused state variable array to counteract their inclusion.

Refer to the `https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#modifying-your-contracts` for more information.

Resolution

The issue has been acknowledged by the development team with the following comment:

> This issue is recognized, and we will subsequently implement a storage gap in all inherited Solidity contracts.

MNT-34	Missing Zero Address Checks In Checkpoint Oracle
Asset	`l2geth/contracts/checkpointoracle/contract/oracle.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The checkpoint oracle contains no zero address checks on admin addresses set by the `constructor()` function.

If the zero address was accidentally set as a valid admin address it would result in any invalid signature being marked as a valid whitelisted address.

Note, this finding is marked as informational as currently this contract does not appear to be used by Mantle.

Recommendations

Include a zero address check if there is any intention of using the checkpoint oracle in future deployments. If the contract is not needed, remove it from the project.

Resolution

The issue has been addressed at PR-1135.

MNT-35	TSS Node Clients Have Unreachable Max Connection Attempts Logic
Asset	`tss/ws/client/tm/client.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The WebSocket client of a TSS node employs an exponentially increasing time delay for reconnect attempts.

When attempting to reconnect the WebSocket client of a TSS node, there is an exponentially increasing time delay to prevent accidental DoS attacks on the server being connected to. If the max number of reconnect attempts is reached then the program should return an error on line [**308**] of `tss/ws/client/tm/client.go`.

However, because the time delay increases exponentially, and the max amount of tries is 25, this will lead to an unrealistic waiting periods and over 2 years prior to the error message on line [**308**] being reached. This effectively makes line [**308**] unreachable.

Recommendations

Reduce the exponential time gaps between reconnect attempts. Currently this is calculated by `(1 << uint(attempt)) * time.Second`.

There is no realistic need to have over 1 year between reconnect attempts and, as such, a linear equation for the time delay seems more fitting.

Resolution

The issue has been addressed at PR-1257.

MNT-36	Irregular Size Of The First Signature In Sequence Will Trigger Panic
Asset	`datalayr-mantle/dl-disperser/aggregator.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

In `flattenSigs()`, if the first signature is of incorrect size, the resulting `sigBytes` array will be created with an incorrect length and may panic with out of bounds error:

```
410 func flattenSigs(sigs [][]byte) []byte {
    //this could change
412 sigLen := len(sigs[0])
    // Decompose sig
414 sigBytes := make([]byte, len(sigs)*sigLen)

416   copy(sigBytes[i*sigLen:], sigs[i])
    for i := 0; i < len(sigs); i++ {
418 }
    return sigBytes
420 }
```

Note, this function is used by `datalayr-mantle/dl-disperser/utils.go` `genRandomSigs()`, which did not appear to be used in the reviewed codebase.

Recommendations

Use the fixed `crypto.SignatureLength` from the package crypto instead of `sigLen := len(sig[0])` on line [**412**].

Resolution

The issue has been acknowledged by the development team with the following comment:

> The code of the old version of Eigenlayr, which is currently obsolete. flattenSigs() has been removed.

MNT-37	Unused Items in Structures
Asset	`mt-batcher/services/sequencer/driver.go`, `mt-batcher/services/restorer/service.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Unused items in structures:

1. `Driver`
   - `GraphqlClient`
2. `DriverConfig`
   - `EigenABI`
   - `EigenFeeABI`
   - `RollupMinSize`
   - `ChainID`
   - `EigenLogConfig`
   - `CheckerBatchIndex`
3. `DaService`
   - `GraphClient`
4. `DaServiceConfig`
   - `EigenABI`
   - `Timeout`
   - `Debug`

Recommendations

Remove all unused elements from the struct.

Resolution

The issue has been addressed at PR-1202.

MNT-38	Miscellaneous General Comments
Asset	`Various files`
Status	**Closed:**
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Semantic overloaded variables**

   On line [**215**] of the smart contract `TssGroupManager` an active member's existence is determined by checking the length of their public key. It is best to avoid adding silent extra meanings to variables as when later changes are made it may not be obvious of their impact to such checks. See the OpenZeppelin forums for more details.

2. **Stale code**

   In various places of the code there is code that is commented out and no longer used, this code should be removed to avoid mistakes in the future. Likewise some contracts were confirmed to no longer be used and so should be removed from the project. Examples are:

   - line [**65**] of `TssGroupManager`,
   - the `WETH9` contract and
   - lines [**127-138**] of `tss/node/tsslib/p2p/communication.go`.

3. **Lack of zero address checks**

   In the smart contract function `TssStakingSlashing.setAddress()` there are no zero address checks on the argument inputs. This could result in one or both contracts being set to the default state if care is not taken whilst calling this function. It is suggested to verify both arguments are set prior to overwriting `bitToken` and `tssGroupContract` in storage.

4. **Use of SafeMath library with wrong Solidity version**

   `TssRewardContract`, `BVM_EigenDataLayrFee`, `BVM_EigenDataLayrChain` and `TssGroupManager` all import the SafeMath library or its upgradable variant despite using the versioning `pragma ^ 0.8.0` or `pragma ^ 0.8.9` in which SafeMath behaviour is included by default.

5. **Incorrect comments in codebase**

   There are several instances of incorrect comments in the code:

   - In `BVM_EigenDataLayrChain` on line [**197**] there is an incorrect revert message for function `restRollupBatchData()`.
   - In `BVM_EigenDataLayrChain` on line [**120**] there is an incorrect revert message for function `setFraudProofAddress()` as this function adds an address to the fraud proof whitelist, it does not remove it.
   - In `BVM_EigenDataLayrChain` on line [**319**] the NatSpec comment is incorrect as currently the challenger role has whitelisted access only.

6. **Magic numbers**

   When constant values are used these should be set as named constant variables rather than hardcoded values within the code body, this aids readability and makes future alterations easier for developers.

   One Golang example is the maximum batch size in `mt-batcher/services/restorer/handle.go` on line [**97**].

   For Solidity examples, on line [**75**] of `TssRewardContract` both `10 ** 18` and `365 * 24 *60 *60` should be stored in named variables. Both could also make use of built-in Solidity number aliases, `1 ether == 10 ** 18` and `52 weeks == 365 * 24 * 60 * 60`.

7. **Numerous TODO comments to address**

   Numerous TODO comments exist throughout the Solidity and Golang code outlining outstanding design decisions and feature implementation. Go through all TODO comments to ensure all key design decisions have been made, verified and there are no outstanding items that could be considered critical.

8. **Explicit `panic()` calls throughout the codebase**

   Number of explicit `panic()` calls exist throughout the Golang code. Investigate each occurrence individually to determine if the error conditions should be handled gracefully, or whether exiting with a panic is appropriate.

9. **Use of depreciated library `ioutil`**

   The `io/ioutil` package has turned out to be a poorly defined and hard to understand collection of things. All functionality provided by the package has been moved to other packages and so should not be used. A non-exhaustive list of uses in the codebase is as follows:

   - `node/tsslib/storage/localstate_mgr.go` uses `ioutil.ReadFile` and `ioutil.WriteFile`.
   - `ws/client/tm/http_json_client.go` uses `ioutil.ReadAll`.
   - `proxyd/backend.go` uses `ioutil.ReadAll`.
   - `proxyd/rpc.go` uses `ioutil.ReadAll`.
   - `proxyd/server.go` uses `ioutil.ReadAll`.
   - `proxyd/tls.go` uses `ioutil.ReadFile`.
   - `subsidy/cache-file/cache.go` uses `ioutil.ReadFile`.
   - `tss/node/tsslib/storage/localstate_mgr.go` uses `ioutil.ReadFile` and `ioutil.WriteFile`.

10. **Use of depreciated error checking method**

    The following files all make use of `os.IsNotExist()` for error detection, this only supports errors returned by the *os* package and so new code should use `errors.Is(err, fs.ErrNotExist)` to detect errors instead.

    - `tss/node/tsslib/storage/localstate_mgr.go`,
    - `tss/node/tsslib/tss.go` and
    - `subsidy/cache-file/cache.go`.

11. **Gas Optimizations**

    Some areas of code can be optimized further, this is important particularly for code intended to run on the Ethereum mainnet due to the transaction cost being generally higher there. Some examples are:

    - Calling a length in a loop, one such example is on line [**67**] of `TssGroupManager`: To save gas this length should be called once and stored in a local variable rather than called every iteration of the loop.
    - Checks against booleans. Rather than checking if `boolean == true` it is better to just check the boolean condition itself. Examples are line [**95**] and line [**98**] of `TssGroupManager`.
    - Accessing storage variables repeatedly. If a variable is needed multiple times in the same function it is better to write the value to a local variable and access that repeatedly. An example of this is `memberGroupKey[_publicKey]` in `TssGroupManager.setGroupPublicKey()` that is accessed 3 times.

- Redundant checks. Some sections of code are unnecessary as shown below:
  - line [**127**] of `TssStakingSlashing` is not needed as if `slashAmount[0] > 0` then `exIncome[0] > 0` also. This is because the values of these variables can only be set via calling `setSlashingParams()` only and line [**103**] and line [**104**] enforce this relationship between `slashAmount[i]` and `exIncome[i]`.
  - `TssGroupManager.removeActiveTssMembers()` validates that the index given is less than the length of the array. However, this function is only ever called by `TssGroupManager.removeMember()` which bounds the input `i` by the array length already, making the check in `removeActiveTssMembers()` redundant.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned in PR-1140 and commit bb394d0.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

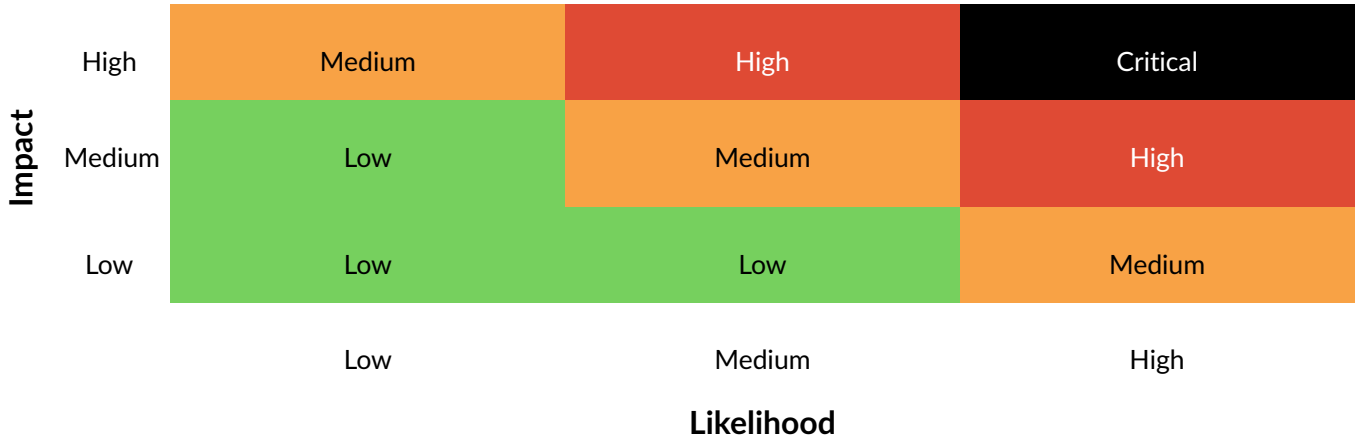

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].