ChainSafe

Gossamer

Security Assessment Report

Version: 1.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Gossamer codebase, a Polkadot Host implementation. The review focused solely on the security aspects of the codebase, 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 software. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model, crypto-economic analysis or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Gossamer software 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*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Gossamer smart contracts.

Overview

Gossamer is an implementation of the Polkadot Host software, written in Go. When provided a compliant web-assembly (WASM) runtime[1] and a genesis specification, it is designed to function as node software that is suitable for chains in the Polkadot network ecosystem — an alternative to nodes built using Substrate.

In addition, Gossamer is designed as a modular framework and library, to enable Go developers to depend on Gossamer in order to implement customised blockchain protocols — similar to what Substrate provides for Rust developers. Gossamer does not currently include any assistance with developing the WASM runtime itself, and this must be created through external means such as with Substrate FRAME.

[1] This defines the blockchain's state transition. Refer to Polkadot Glossary

Security Assessment Summary

This review was conducted on the files hosted in the Gossamer repository, which were assessed at tag v0.6.0.

The following were used for comparison and reference but were not directly in scope of the assessment:

- `w3f/polkadot-spec` : tag v0.1.0 — the most recent release upon commencement of the review.
- `paritytech/polkadot` : commit 4c3166c.
- `paritytech/substrate` : commit fb08d15, and dependencies described in the project's `Cargo.lock` file.

The testing team performed a time-boxed, manual security review of the Gossamer codebase to identify issues and vulnerabilities that could be caused by unexpected behaviour, malicious messages, unusual network conditions, and behaviour inconsistent with the Substrate reference implementation.

In addition to this manual review, the testing team leveraged white-box fuzz testing techniques (i.e. fuzzing). This is a process that enables the identification of bugs by providing mutated and coverage–guided data inputs to software with the purpose of exercising as many code paths as possible, triggering crashes (or in Golang, *panics*) and other unexpected behaviours (e.g. memory leaks). By writing *fuzzing harnesses* that crash when relevant properties and invariants are violated, it is possible perform property-based testing, to gain some probabilistic assurances that the code under test abides by these properties.

The testing team also performed some *differential fuzzing*, implementing harnesses that exercise the same functionality in the Gossamer and Substrate codebases and verify that the output is equivalent.

Along with fuzzing activities, the testing team performed tailored and targeted testing to verify functionality and identify some of the vulnerabilities raised in this report. The relevant scripts and test vectors have also been shared with the Gossamer development team. Output is detailed in Appendix A.

Review Coverage

Due to the time-boxed nature of the review, the following components were subject to limited examination:

- JSONRPC implementation — this was reviewed for resistance to malicious input, but was not evaluated for correctness when compared to the Substrate implementation and the draft standard PSP-06.
- Networking stack and resistance to network-level abuse by malicious peers — this was not reviewed in detail.

Findings Summary

The testing team identified a total of 24 issues during this assessment. Categorised by their severity:

- Critical: 6 issues.
- High: 4 issues.
- Medium: 5 issues.
- Low: 6 issues.
- Informational: 3 issues.

Recommendations

Sigma Prime recommends further, substantial review upon remediation of the issues detailed in this report. This is due to:

- The substantial code changes likely required to remediate some issues will result in significant parts of the software lacking review.
- The severe nature of some of these issues, the time-boxed nature of this review, and lack of exhaustive testing, the team is not confident that other issues are absent.
- The testing team did not perform a sufficient review of the networking layer to be confident in its safety.

In addition, the testing team recommends:

- Appropriate network stress testing, demonstrating resilience when under attack, and recovery after extended periods of non-finality.
- More exhaustive comparative testing to verify consistent functionality across the various Polkadot Host implementations. This should ideally include differential fuzzing, or similar.
- A more explicit, concrete specification that includes type information (where relevant to consensus). Optimally, it should describe an expected range of constants and runtime values under which a compliant Polkadot Host should function.

  It is not obvious what Substrate functionality must be implemented by a compliant Polkadot host, and which is additional functionality provided by the Substrate framework (in excess of that required).

The testing team indicates that remediation of the issues detailed in this report, alone, is not sufficient for production use.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Gossamer codebase. 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
GSR-01	Forks in With Different BABE and GRANDPA Digest Messages Overwrite State	Critical	Open
GSR-02	Forks in Blocks with Different GRANDPA Digests Result in a Consensus Split	Critical	Open
GSR-03	GRANDPA Does Not Verify Equivocatory Votes	Critical	Open
GSR-04	Block Numbers and Not Verified Against Block Hashes in GRANDPA	Critical	Open
GSR-05	Insufficient Garbage Collation for GRANDPA Message Tracker	Critical	Open
GSR-06	Denial-of-Service Condition via Decoding of Malicious Messages	Critical	Open
GSR-07	GRANDPA Does Not Check Multiplicity of Equivocatory Votes	High	Open
GSR-08	Race Condition May Result in Mismatch of `Runtime` and `TrieState`	High	Open
GSR-09	Incorrect Bounds Checking Skips Verification of Epoch 0	High	Open
GSR-10	Panic When Deleting Nonexistent Keys From Trie	High	Open
GSR-11	Duplicate Votes in GRANDPA are Counted as Equivocatory Votes	Medium	Open
GSR-12	Multiple Different Schedule Changes Are Handled Differently to Substrate	Medium	Open
GSR-13	Storing Justification Allows Excess Bytes	Medium	Open
GSR-14	Unsafe Type Casting from Untrusted Input	Medium	Open
GSR-15	Assumption of JSON Fields from Untrusted Input	Medium	Open
GSR-16	Specification Issues and Limitations	Low	Open
GSR-17	`VerifyBlockJustification()` Does Not Verify Justification Hash Relates to Block Being Justified	Low	Open
GSR-18	Casting of `uint32` to `int` Will Overflow In 32-bit OS	Low	Open
GSR-19	Unrestricted Loop When Building Blocks	Low	Open
GSR-20	Insecure Security Disclosure Policy	Low	Open
GSR-21	Incorrect Use of Synchronisation Primitives	Low	Open
GSR-22	BABE uses Milliseconds Rather Than Nanoseconds	Informational	Open
GSR-23	Incorrect Log for `SealDigest` Verification	Informational	Open
GSR-24	Miscellaneous Gossamer Issues	Informational	Open

GSR-01	Forks in With Different BABE and GRANDPA Digest Messages Overwrite State		
Asset	`dot/state/epoch.go` & `dot/state/grandpa.go`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The state storage structures `EpochState` and `GrandpaState` are key-value databases which are not resilient to forks. Forks may occur when importing unfinalised blocks. The keys used to store data in `EpochState` and `GrandpaState` will overlap on forks and thus overwrite the data for other blocks.

An example of this issue can be seen in BABE, a Proof of Stake (PoS) protocol which acts as a block production engine. To facilitate PoS there is an authority list for each epoch, which specifies which nodes are potential block producers. The authority list is updated on the first block of the previous epoch.

On the first block of each epoch, the runtime generates a digest item `NextEpochData`, which tells BABE the authority list for the next epoch. It is stored in the `EpochState` via the `SetEpochData(epoch uint64, info *types.EpochData)` function. This will store the new epoch data in the key-value database using the key as `epochPrefix || epoch` and data as the `EpochData`.

An issue arises if there is a fork with two different blocks being processed immediately after an epoch boundary (i.e. different first blocks of an epoch). If these blocks generate a different `NextEpochData` digest message they will both have the same key `epochPrefix || epoch`, as `epoch` is just one more than the current epoch. The implications occur on insertion into the database, where the keys will match, hence the value (i.e. `EpochData`) will be overwritten.

As an example consider the following situation. Say there are 60 slots in an epoch and the blocks are as follows:

- blockA: slot 59
- blockB: slot 60, parent blockA
- blockC: slot 61, parent blockA
- blockD: slot 62, parent blockB

`NextEpochData` will be updated in the first slot of each epoch, due to the fork this will be `blockB` and `blockC`. If the blocks are processed in alphabetical order (A, B, C, D) then `EpochState` will store `EpochData` generated in `blockC` rather than `blockB` and thus children of `blockD` will incorrectly use the `EpochData` from `blockC`.

The impact is that nodes processing these blocks in a different order (e.g. A,B,C,D vs A,B,D) will have a different authority list and cause a consensus split. The split will not be mitigated until the nodes clears their databases and imports just the canonical blocks.

The same issue arises in the function `SetConfigData(epoch uint64, info *types.ConfigData)`, which also only uses `epoch` as a key, thus will not handle forks.

A similar issue is present in `GrandpaState`, as the keys used are based on either the `SetID` or `Number` (block number) neither of which will be unique over forks.

Updating GRANDPA state to account for forks is required to correctly determine when the `NextGrandpaAuthorityChange()` will occur. This is used by `determinePreVote()` and `determinePreCommit()` in `lib/grandpa/grandpa.go`. If authority changes occur at different times this will impact which block may be voted for.

Recommendations

Consider storing in memory the current `EpochState` and `GrandpaState` for each block. Then store on disk the `EpochState` and `GrandpaState` for finalised blocks. States related to blocks strictly earlier than the last finalised may be pruned from memory (nodes may be imported with the parent as the last finalised block, thus these states are still required in memory). When a new block is imported the correct states may be fetched by using the parent hash of the new block.

Note that, by storing states in memory, all blocks that are not finalised will need to be imported again after shutdown in order to generate the states in memory.

GSR-02	Forks in Blocks with Different GRANDPA Digests Result in a Consensus Split	
Asset	`dot/digest/digest.go`	
Status	**Open**	
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The GRANDPA protocol is a finality gadget which allows a modifiable authority list to vote on a which blocks should be finalised. The authority list may be changed after a set delay which is triggered by digest messages received by the runtime.

There is only one `Handler` which processes all blocks, on any fork. The `Handler` has fields relating to GRANDPA changes which are not updated if we switch to another fork. As a result the `Handler` will have incorrect values from the GRANDPA changes when we switch forks.

See the definition of `Handler` in the code snippet below for the list of variables required for GRANDPA changes.

```go
type Handler struct {
  // ...
  // GRANDPA changes
  grandpaScheduledChange *grandpaChange
  grandpaForcedChange    *grandpaChange
  grandpaPause           *pause
  grandpaResume          *resume
}
```

The issue can be seen in the handling of any GRANDPA digest messages, `handleScheduledChange()` will be used as an example as seen below.

```
func (h *Handler) handleScheduledChange(sc types.GrandpaScheduledChange, header *types.Header) error {
    curr, err := h.blockState.BestBlockHeader()
    if err != nil {
        return err
    }

    if h.grandpaScheduledChange != nil {
        return nil
    }

    logger.Debugf("handling GrandpaScheduledChange data: %v", sc)

    c, err := newGrandpaChange(sc.Auths, sc.Delay, curr.Number)
    if err != nil {
        return err
    }

    h.grandpaScheduledChange = c

    auths, err := types.GrandpaAuthoritiesRawToAuthorities(sc.Auths)
    if err != nil {
        return err
    }
    logger.Debugf("setting GrandpaScheduledChange at block %s",
        big.NewInt(0).Add(header.Number, big.NewInt(int64(sc.Delay))))
    return h.grandpaState.SetNextChange(
        types.NewGrandpaVotersFromAuthorities(auths),
        big.NewInt(0).Add(header.Number, big.NewInt(int64(sc.Delay))),
    )
}
```

Consider the following fork.

- blockA: slot 10
- blockB: slot 11, parent blockA
- blockC: slot 12, parent blockA
- blockD: slot 13, parent blockC

Assume `blockB` and `blockC` have different schedule change digest messages. When `blockB` is processed `grandpaScheduledChange` will be set to that generated by the digest message in `blockB`. However, when `blockC` is processed there is already data in `grandpaScheduledChange` thus we will trigger the following conditional statement, ignoring the scheduled change in `blockC`.

```
if h.grandpaScheduledChange != nil {
    return nil
}
```

Thus, any children of `blockC` will have the wrong authority list.

The impact is a consensus split as the authority list will be different for nodes depending on what order they processed the blocks (e.g. A,B,C,D vs A,C,D), which may lead to a loss of finality on networks with a significant proportion of authorities running Gossamer.

Recommendations

The GRANDPA changes should be maintained separately for each fork. Consider adding these to the `GrandpaState` and implement the solution recommended in GSR-01.

GSR-03	GRANDPA Does Not Verify Equivocatory Votes		
Asset	`lib/grandpa/message_handler.go`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

An equivocatory vote occurs when a validator has two or more votes at the same stage during a single round. Such as if a validator makes two pre-votes in round 10 with set ID 1.

In the `verifyCommitMessageJustification()`, `verifyPreCommitJustification()` and `VerifyBlockJustification()` functions, equivocatory voters are found through the `getEquivocatoryVoters()`. This searches through the `[]AuthData` and records each `AuthorityID` (i.e. public key) that occurs more than once in the list.

If a voter has been recorded as an equivocatory voter then their vote will not be verified. The specifications state that if any of the conditions are not met then a vote is invalid.

- The block hash does not correspond to a valid block.
- The block is not an (eventual) descendant of a previously finalized block.
- The signature is invalid.
- The AuthorityID is not in the authority list for the set.

A vote is required to be valid to be counted as an equivocatory vote. Equivocatory votes are still counted towards reaching the threshold for finalisation, noting that they are given a weight of one vote, rather than taking the total number of votes that authority has equivocated.

The impact of this is that we are able to bypass verification checks by posting equivocatory votes.

For example, bypassing signature verification allows an attacker to make multiple votes from authorities without their private key. As a result the attacker is able to forge equivocatory votes for any arbitrary block which count towards the threshold and thus they are able to finalise any arbitrary block.

Bypassing authority list checks allows the attacker to generate as many public keys as required and sign multiple votes which will be counted as equivocatory votes for an arbitrary block.

One example of the issue is seen in the code snippet below for `verifyPreCommitJustification()`. Here, when a voter is in the `eqvVoters` map, no more verification will occur. However, they are still counted in the comparison against the threshold `if count + uint64(len(eqvVoters)) < h.grandpa.state.threshold()`.

```go
func (h *MessageHandler) verifyPreCommitJustification(msg *CatchUpResponse) error {
  auths := make([]AuthData, len(msg.PreCommitJustification))
  for i, pcj := range msg.PreCommitJustification {
    auths[i] = AuthData{AuthorityID: pcj.AuthorityID}
  }

  eqvVoters := getEquivocatoryVoters(auths)

  // verify pre-commit justification
  var count uint64
  for idx := range msg.PreCommitJustification {
    just := &msg.PreCommitJustification[idx]

    if _, ok := eqvVoters[just.AuthorityID]; ok {
      continue
    }

    err := h.verifyJustification(just, msg.Round, msg.SetID, precommit)
    if err != nil {
      continue
    }

    if just.Vote.Hash == msg.Hash && just.Vote.Number == msg.Number {
      count++
    }
  }

  if count+uint64(len(eqvVoters)) < h.grandpa.state.threshold() {
    return ErrMinVotesNotMet
  }

  return nil
}
```

Recommendations

Ensure each equivocatory vote is valid for the functions `verifyCommitMessageJustification()`, `verifyPreCommitJustification()` and `VerifyBlockJustification()` otherwise disregard the vote.

GSR-04	Block Numbers and Not Verified Against Block Hashes in GRANDPA		
Asset	`lib/grandpa/message_handler.go`		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

GRANDPA messages such as votes, commits and justifications often have both a `Hash` field and `Number` field. Where `Hash` is the block hash and `Number` is the block number associated with the hash.

Gossamer does not verify that the `Number` matches the `Hash`.

The following functions do not verify a `Number` is associated with a `Hash`:

1. `VerifyBlockJustification()` does not verify the `Justification.Commit.Number` against `Justification.Commit.Hash`;
2. `VerifyBlockJustification()` does not verify the `GrandpaVote.Number` against `GrandpaVote.Hash` for each commit in `Justification.Commit.Percommits`;
3. `handleCatchUpResponse()` does not verify `CatchUpResponse.Number` against `CatchUpResponse.Hash`;
4. `handleCommitMessage()` does not verify `CommitMessage.Vote.Number` against `CommitMessage.Vote.Number`;
5. `verifyPreCommitJustification()` does not verify `CatchUpResponse.PreCommitJustification.Vote.Number` against `CatchUpResponse.PreCommitJustification.Vote.Number`;
6. `verifyPreVoteJustification()` does not verify `CatchUpResponse.PreCommitJustification.Vote.Number` against `CatchUpResponse.PreCommitJustification.Vote.Number`;
7. `handleCommitMessage()` does not verify `CommitMessage.Vote.Number` against `CommitMessage.Vote.Number`;

The impact here is that we may have malformed justifications used in finalisations, causing a consensus split with Substrate.

Furthermore, justifications for a block are not overwritten once they are set and so will remain until we clear the database and resync.

Additionally, this will cause issue when calculating the GHOST and sorting blocks by their number in `getPreVotedBlock()`, which may lead to precommitting to an invalid block.

Recommendations

This issue may be resolved by adding validation checks for each of these messages to ensure that the `Number` matches the `Hash`. The checks may be implemented by using the `Hash` to fetch the block header from the block state. Then ensuring the `Number` equals the number in the block header.

Note that items *5-7* may be resolved by handing these checks in `validateJustification()`.

GSR-05	Insufficient Garbage Collation for GRANDPA Message Tracker		
Asset	`lib/grandpa/vote_message.go` & `lib/grandpa/message_tracker.go`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In the GRANDPA protocol, if vote or commit messages are received which are unable to be processed, the related block is added to the `tracker` for future processing. This may occur due to a block not existing in the block tree or that the round in a vote message is greater than the current round. Future processing is triggered when a new block is imported, the `tracker` then compares the block hash of the new block against the hash included in the messages. When the hashes match, the message is passed back to the handler for processing.

There is a potential denial of service (DoS) attack that can exploit the storage of these messages. When a vote is received with a round greater than our current round it will be added to the `tracker`. An attacker is able to exploit this by repeatedly sending vote messages with rounds much higher than the current round. These votes will be stored in the `tacker` indefinitely if the hash is not from a block that will not be imported (e.g. a hash of random bytes). Thus, memory will continue to grow indefinitely as the attacker continues to send malicious votes with high rounds. Each round must be different else it is an equivocatory vote, however `round` is a `uint64` which gives 2^{64} different rounds to use.

Furthermore, if there are valid votes received from a future round while our round is behind and we have imported the block associated with the `Hash`, then these votes will be added to the `tracker` and never processed. This is due to the fact we only iterate through the `tracker` when we import a new block, checking the vote hash against the new block hash. Hence, votes for blocks already in the tree will never be processed.

The votes or commits which are stored in the `tracker` will be stored indefinitely when they have a `Hash` related to blocks that will never be imported. Since the commit messages are validated and thus will be related to finalised blocks, it can be assumed that all commit messages will be processed eventually as all finalised blocks will be imported. However, due to race conditions between storing the messages in the maps and blocks being imported, it is possible to have commit messages included in the maps which relate to blocks already imported. Furthermore, a malfunctioning or malicious node may send a vote with a block hash not related to a real block. These block hashes will not be processed and will remain in the `tracker` indefinitely.

Recommendations

The issue may be partially resolved by ignoring votes with the `Round` higher than our current `Round`. Valid nodes will periodically resend votes for the duration of a round, thus the current round votes will eventually be received. This can be implemented in `lib/grandpa/vote_message.go::validateMessage()`.

We also recommend setting an expiry time for vote and commit messages that are stored in the `tracker`. A garbage collector routine may then be used to periodically iterate over the `voteMessages` and `commitMessages` maps and delete those that have expired.

GSR-06	Denial-of-Service Condition via Decoding of Malicious Messages		
Asset	`pkg/scale/decode.go`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A malicious node can readily cause all connected Gossamer peers to crash by broadcasting maliciously crafted messages, without requiring significant expenditure of resources.

SCALE messages from untrusted sources are decoded without appropriately validating length values, restricting them to appropriate maximums. Large length values result in an attempt to allocate excessive memory, triggering an unrecoverable "out of memory" (OOM) crash.[2]

As malformed or malicious messages from unauthenticated sources can cause all receiving Gossamer nodes to crash, this has been deemed of *critical* severity. Such an attack requires little expertise (sending random data is likely to trigger this) and would have a significant impact on the health of any network running a suitable proportion of Gossamer nodes.

Detail

This issue was identified via fuzzing. The relevant crashes were triggered within `(*decodeState).decodeBytes()` at line [525] (shown below). The decoded length value is directly passed to `make()`, resulting in an attempt to allocate sufficient memory and triggering the crash.

```
518  // decodeBytes is used to decode with a destination of []byte or string type
     func (ds *decodeState) decodeBytes(dstv reflect.Value) (err error) {
520    length, err := ds.decodeLength()
       if err != nil {
522      return
       }
524
       b := make([]byte, length)
526    // ...
```

As the value returned by `(*decodeState).decodeLength()` does not appear to be validated in the other places it is used, similar crashes can also be expected there.

Any associated crashes are triggered before other validation e.g. signature verification.

Relevant fuzzing crashes were identified when decoding SCALE data into the following types:

- `BlockAnnounceMessage`
- `BlockResponseMessage`
- `LightRequest`
- `LightResponse`

[2]It executes an unrecoverable runtime.throw

- `VersionData`
- `LegacyVersionData`
- `NewBodyFromBytes`

Of particular concern are messages that are broadcast to all peers, like `BlockAnnounceMessage` .

Recommendations

Validate length values within all SCALE-encoded objects received from external sources.

The testing team recommend that the Polkadot Host specification be extended to define appropriate maximum length limits for all fields of variable length transmitted via the network. Compliant implementations should enforce these limits.

Additionally, SCALE does not involve compression so decoded message content (without length info) should be shorter than the encoded message (containing length info). As such, it should be impossible for any valid SCALE-encoded data to contain a length value larger than total length of the data. This could be helpful for validating length values.

If validation is expensive, it could be reasonable to avoid length validation when decoding trusted, internal data (e.g. from the database). If so, it is crucial that all data received from untrusted sources are clearly separated from trusted data. One solution could involve enforcing this via the type system e.g. a `BlockResponseMessage` [3] contains a sequence of `types.BlockData` [4], but the BlockDB stores data of type `TrustedBlockData` , which is decoded using `scale.UnsafeUnmarshal()` , with unchecked lengths.

[3] NOTE: while a BlockResponseMessage is encoded using protobuf, fields within the message are SCALE encoded to byte arrays, so this type is relevant.

[4] defined in `dot/network/message.go:207`

GSR-07	GRANDPA Does Not Check Multiplicity of Equivocatory Votes		
Asset	`lib/grandpa/message_handler.go`		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

An equivocatory vote occurs when a validator has two or more votes at the same stage during a single round.

The specifications state that a valid justification must not contain more than two equivocatory votes from each voter.

The Gossamer function `getEquivocatoryVoters()` does not put a restriction on the number of equivocatory votes from each voter.

As a result, commits that may be rejected in Substrate could be accepted in Gossamer. Furthermore, it is an inefficient use of space and bandwidth to include more than two equivocatory votes.

Recommendations

The issue can be mitigated by rejecting commits with more than two equivocations per user.

This can be implemented by having `getEquivocatoryVoters()` return an error if any authority has more than two equivocations as seen below.

```go
func getEquivocatoryVoters(votes []AuthData) (map[ed25519.PublicKeyBytes]struct{}, error) {
    eqvVoters := make(map[ed25519.PublicKeyBytes]struct{})
    voters := make(map[ed25519.PublicKeyBytes]int, len(votes))

    for _, v := range votes {
        voters[v.AuthorityID]++

        if voters[v.AuthorityID] > 2 {
            return nil, errors.New("Invalid Multiplicity for Voter")
        }

        if voters[v.AuthorityID] == 2 {
            eqvVoters[v.AuthorityID] = struct{}{}
        }
    }

    return eqvVoters, nil
}
```

GSR-08	Race Condition May Result in Mismatch of `Runtime` and `TrieState`		
Asset	`dot/core/service.go`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There is a concurrency issues related to fetching `TrieState(nil)` and `GetRuntime(nil)` separately, which may result in different blocks being used for the `TrieState` and `Runtime`. The impact is significant as the `TrieState` stores the execution code for a runtime and thus may result in an invalid bytecode being used by the runtime.

The function `HandleSubmittedExtrinsic()` is an RPC call that runs in parallel to block importation and has the following code.

```go
// HandleSubmittedExtrinsic is used to send a Transaction message containing a Extrinsic @ext
func (s *Service) HandleSubmittedExtrinsic(ext types.Extrinsic) error {
  if s.net == nil {
    return nil
  }

  ts, err := s.storageState.TrieState(nil)
  if err != nil {
    return err
  }

  rt, err := s.blockState.GetRuntime(nil)
  if err != nil {
    logger.Critical("failed to get runtime")
    return err
  }

  rt.SetContextStorage(ts)
// ...
```

The issue here is that passing `nil` to `s.blockState.GetRuntime(nil)` and `s.storageState.TrieState(nil)` tells the block state to use the best block. However, it is possible for the best block to be updated between these calls by the block handler routine which may import a new block. Thus, `ts` may `rt` may point to different blocks.

Recommendations

We recommend first fetching the best block hash then passing this hash to each function. See the code snippet below for a potential implementation.

```go
// HandleSubmittedExtrinsic is used to send a Transaction message containing a Extrinsic @ext
func (s *Service) HandleSubmittedExtrinsic(ext types.Extrinsic) error {
  if s.net == nil {
    return nil
  }

  bestBlockHash, err := s.blockState.BestBlockHash()
    if err != nil {
    return err
  }

  ts, err := s.storageState.TrieState(bestBlockHash)
  if err != nil {
    return err
  }

  rt, err := s.blockState.GetRuntime(bestBlockHash)
  if err != nil {
    logger.Critical("failed to get runtime")
    return err
  }

  rt.SetContextStorage(ts)
// ...
```

GSR-09	Incorrect Bounds Checking Skips Verification of Epoch 0		
Asset	`dot/state/epoch.go`		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When importing blocks from file we are able to skip verification to speed up the import time.

The function `SkipVerify()` as seen below, will skip BABE verification if the epoch is less than or equal to `skipToEpoch`.

When running a production node, the variable `skipToEpoch` should not be used, however it will default to 0. Thus, headers will never be verified for epoch 0. As epochs begin counting from 0 the first epoch worth of blocks will not be verified allowing nodes to send malicious blocks that will be accepted.

```
func (s *EpochState) SkipVerify(header *types.Header) (bool, error) {
  epoch, err := s.GetEpochForBlock(header)
  if err != nil {
    return false, err
  }

  if epoch <= s.skipToEpoch {
    return true, nil
  }

  return false, nil
}
```

Recommendations

Consider updating the condition to use a strict inequality (i.e. `<`) as seen below.

```
func (s *EpochState) SkipVerify(header *types.Header) (bool, error) {
  epoch, err := s.GetEpochForBlock(header)
  if err != nil {
    return false, err
  }

  if epoch < s.skipToEpoch {
    return true, nil
  }

  return false, nil
}
```

GSR-10	Panic When Deleting Nonexistent Keys From Trie		
Asset	`lib/trie/trie.go`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Under some circumstances, deleting a key from a trie triggers a `slice bounds out of range` runtime panic, causing the Gossamer node to crash.

In particular, deleting a nonexistent key k_x will raise a panic when the trie contains valid key k_y for which k_x is a prefix i.e. where `bytes.HasPrefix(k_y, k_x)` [5] holds.

While this cannot be exploited by network messages from unauthenticated nodes, it may be readily possible for the runtime to trigger this crash; in particular via the `ext_storage_clear` and `ext_default_child_storage_clear` Host API functions, which may be possible for unprivileged users to execute in some runtimes. Malicious block proposers may also be able to directly exploit this.

Once a block contains such an operation, it becomes impossible for Gossamer nodes to execute the state transition without crashing — resulting in repeated crashes and possibly a chain split until the issue is patched.

Detail

This issue was identified via fuzzing, and is triggered in the `(*Trie).delete()` function, listed below.

```
703  func (t *Trie) delete(parent node, key []byte) (node, bool) {
       // Store the current node and return it, if the trie is not updated.
705    switch p := t.maybeUpdateGeneration(parent).(type) {
         case *branch:
707
         length := lenCommonPrefix(p.key, key)
709      if bytes.Equal(p.key, key) || len(key) == 0 {
           // found the value at this node
711        p.value = nil
           p.setDirty(true)
713        return handleDeletion(p, key), true
         }
715
         n, del := t.delete(p.children[key[length]], key[length+1:])
717      if !del {
           // If nothing was deleted then don't copy the path.
719        return p, false
         }
```

In a relevant fuzzing crash, `lenCommonPrefix()` at line [**708**] was called with arguments:

```
p.key := []byte([6, 1, 6, 2, 6, 3, 2, 13])
key := []byte([6, 1, 6, 2, 6, 3])
```

[5] https://pkg.go.dev/bytes#HasPrefix

And returning a value of `length := 6`.

A `slice bounds out of range [7:6]` runtime error is raised at line [716] after attempting to create a sub-slice of `key[7:]`, when `len(key) == 6`.

Recommendations

The testing team recommends interfacing with the maintainers of the Host Specification and reference implementation — to clarify and explicitly specify whether a compliant runtime should ever call `ext_storage_clear` and `ext_default_child_storage_clear` with nonexistent keys and, if so, what should occur.

If, under those circumstances, `(*Trie).Delete()` should be effectively a no-op, properly ensure that it can handle nonexistent key arguments without panicking.

Solutions could involve:

- Checking that the value `length + 1` is within the bounds of the `key` slice i.e. that `length < len(key)`.
- Handling the specific case where the parent key `p.key` is longer than the `key` argument.

GSR-11	Duplicate Votes in GRANDPA are Counted as Equivocatory Votes		
Asset	`lib/grandpa/message_handler.go`		
Status	Open		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The function `getEquivocatoryVoters()` is used to determine if a voter has equivocated by voting twice in a single round of GRANDPA. The function can be seen in the following code snippet.

```
func getEquivocatoryVoters(votes []AuthData) map[ed25519.PublicKeyBytes]struct{} {
  eqvVoters := make(map[ed25519.PublicKeyBytes]struct{})
  voters := make(map[ed25519.PublicKeyBytes]int, len(votes))

  for _, v := range votes {
    voters[v.AuthorityID]++

    if voters[v.AuthorityID] > 1 {
      eqvVoters[v.AuthorityID] = struct{}{}
    }
  }

  return eqvVoters
}
```

It does not account for the case where a vote appears twice in the `AuthData` list. As a result, an authority may be incorrectly judged as an equivocator when an attacker includes the vote twice.

Recommendations

Duplicates are currently ignored in the Substrate implementation and should be treated the same in Gossamer to avoid a consensus split.

However, it seems beneficial to both protocols to reject commits which have malicious or wasteful data to reduce database size and network bandwidth. Consider updating the specifications to reject commits which include invalid or duplicate votes.

GSR-12	Multiple Different Schedule Changes Are Handled Differently to Substrate		
Asset	`dot/digest/digest.go`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When digest messages are received from the runtime the `Handler` processes these messages and updates the BABE and GRANDPA states respectively.

There is an edge case where, if the runtime returns both a GRANDPA forced change and scheduled change in the same block, Substrate will only processes the forced change. However, Gossamer will process both the scheduled change and forced change.

The Substrate code can be seen below taken from here.

```
  fn check_new_change(
    &self,
  header: &Block::Header,
  hash: Block::Hash,
  ) -> Option<PendingChange<Block::Hash, NumberFor<Block>>> {
    // check for forced authority set hard forks
  if let Some(change) = self.authority_set_hard_forks.get(&hash) {
    return Some(change.clone())
  }

  // check for forced change.
  if let Some((median_last_finalized, change)) = find_forced_change::<Block>(header) {
    return Some(PendingChange {
      next_authorities: change.next_authorities,
      delay: change.delay,
      canon_height: *header.number(),
      canon_hash: hash,
      delay_kind: DelayKind::Best { median_last_finalized },
    })
  }

  // check normal scheduled change.
  let change = find_scheduled_change::<Block>(header)?;
  Some(PendingChange {
    next_authorities: change.next_authorities,
    delay: change.delay,
    canon_height: *header.number(),
    canon_hash: hash,
    delay_kind: DelayKind::Finalized,
  })
}
```

If this case occurs in the runtime, it may trigger a consensus split, depending on if the forced change or scheduled change is implemented.

Recommendations

Consider updating one implementation and the specifications to handle this edge case.

GSR-13	Storing Justification Allows Excess Bytes		
Asset	`dot/sync/chain_processor.go` & `lib/grandpa/message_handler.go`		
Status	Open		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Justifications are sent as raw bytes over the network which are then decoded when they are verified in `VerifyBlockJustification()`.

They are decoded using SCALE decoding, which allows excess bytes to be appended to the end of the data being decoded.

After verification, the original bytes are stored in the block state as seen in `handleJustification()`.

```go
func (s *chainProcessor) handleJustification(header *types.Header, justification []byte) {
  if len(justification) == 0 || header == nil {
    return
  }

  err := s.finalityGadget.VerifyBlockJustification(header.Hash(), justification)
  if err != nil {
    logger.Warnf("failed to verify block number %s and hash %s justification: %s", header.Number, header.Hash(), err)
    return
  }

  err = s.blockState.SetJustification(header.Hash(), justification)
  if err != nil {
    logger.Errorf("failed tostore justification: %s", err)
    return
  }

  logger.Infof("↗ finalised block number %s with hash %s", header.Number, header.Hash())
}
```

This poses a threat, as a malicious actor may append garbage bytes to the end of the justification. These bytes will then be stored in the block state. This is an attack vector for a user who may unnecessarily fill up disk space on a node with garbage bytes.

Recommendations

The Substrate implementation will decode then encode the justification and store the encoded bytes to ensure excess bytes are not included. Consider taking this approach or alternatively rejecting justifications with excess bytes appended as valid nodes will not include these.

GSR-14	Unsafe Type Casting from Untrusted Input		
Asset	`dot/rpc/subscription/websocket.go`		
Status	Open		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Websocket RPC messages are unmarshalled from JSON input before being processed.

After the JSON has been unmarshalled as a `map[string]interface{}` the fields are converted to their required types as seen in the following code block.

```
mbytes, msg, err := c.readWebsocketMessage()
if errors.Is(err, errCannotReadFromWebsocket) {
    return
}

if errors.Is(err, errCannotUnmarshalMessage) {
    c.safeSendError(0, big.NewInt(InvalidRequestCode), InvalidRequestMessage)
    continue
}

params := msg["params"]
reqid := msg["id"].(float64)
method := msg["method"].(string)
```

It is unsafe to assume data decoded from the websockets has the correct type as incorrect types will panic without using safe casting.

For example a malicious user could send a JSON message with `id` field as a `string` then `reqid := msg["id"].(float64)` will panic.

Recommendations

We recommend doing safe, checked casting for all untrusted input. An example can be seen below.

```
reqid, ok := msg["id"].(float64)
if !ok {
    // Handle Error
}
method, ok := msg["method"].(string)
if !ok {
    // Handle Error
}
```

Note: see GSR-15 for further additions to this code.

The `params` value also requires safe casting, however this must be done in each of the RPC methods individually when the type of `params` is known, such as `initExtrinsicWatch()`.

GSR-15	Assumption of JSON Fields from Untrusted Input		
Asset	`dot/rpc/subscription/websocket.go`		
Status	Open		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Websocket RPC messages are unmarshalled from JSON into a map, which should have certain fields.

The following code is taken from `HandleComm()`. Here we assume each of the fields `"params"`, `"id"` and `"method"` exist in the JSON blob.

```
params := msg["params"]
reqid := msg["id"].(float64)
method := msg["method"].(string)
```

If the fields are not present then `nil` will be returned. This may lead to unexpected `nil` pointer exceptions being raised when these pointers are used (for example during `msg["id"].(float64)` when we cast the value to a float).

Recommendations

The issue may be resolved by first ensuring each of the required fields are present in the map before accessing them.

```
var reqid float64
if val, ok := msg["id"]; ok {
    if reqid, ok := val.(float64); !ok {
        // handle error
        return
    }
} else {
    // handle error
    return
}

var method string
if val, ok := msg["method"]; ok {
    if method, ok := val.(string); !ok {
        // handle error
        return
    }
} else {
    // handle error
    return
}
```

GSR-16	Specification Issues and Limitations		
Asset	Polkadot Host Specification `v0.1.1`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Gossamer is designed to implement the Polkadot Host Specification (here referred to as "the spec") and be compatible with the reference implementation built in Rust using Substrate. Consistent functionality is crucial to avoid consensus splits across implementations (where the implementations follow different forks of the chain). Success is hampered, however, when several areas of the specification are lacking detail or missing entirely.

Without an explicit specification, it becomes more difficult to differentiate between parts of the reference implementation that are consensus–critical and those that are unimportant, internal details. When details are unspecified, Gossamer developers must refer directly to Substrate and may inadvertently re-implement bugs, limiting the resiliency afforded by having multiple host implementations.

The following issues relevant to Gossamer were identified that do not directly relate to other issues:

1. The spec should be explicit regarding how the decoding operation should handle situations where extra data is at the end of some SCALE encoded blob. That is, whether decoding is lenient and returns without error, or strict and returns an error.

   This relates to GSR-13.

   This is illustrated in the following fuzzing result, provided alongside this report `fuzz_build/decodeandencodegrandpavoters/crashers/76fa5a5caddef926d4554be1c632e6ab8a04f1fa`

   A strict decoding behaviour is recommended in the context of blockchain node operation.

2. The spec, in particular section 2.1, makes no mention of whether the Storage Trie should allow empty values and empty keys, and how they should be handled.

   In particular Definition 2.4 doesn't clarify whether `storage_set(k, "")` and `storage_clear(k)` are equivalent operations.

   The specification appears to imply that empty keys and values are allowed, and both Gossamer and the Substrate reference implementation abide by this, but an explicit definition is preferred.

3. Section 6.1.1, Remark 6.2 states that "In Polkadot, all authorities have the weight $w_A = 1$", but the threshold in Definition 6.12 simply uses the number of authorities instead of a sum of their weights. In this regard, Gossamer follows the spec but Substrate reads the weights from the runtime.

   substrate:client/consensus/babe/src/authorship.rs:46

   ```
   let theta = authorities[authority_index].1 as f64 /
   authorities.iter().map(|(_, weight)| weight).sum::<u64>() as f64;
   ```

   The main impact is that this could cause a consensus split should a runtime upgrade set authority weights other than 1.

4. Definition 6.12 is insufficient in clarifying how these non-integer computations should be performed - Doesn't specify floating points or how it should be implemented - so rounding and inconsistent results might cause a consensus split (i.e. when the rounding occurs) - Is this floating point? fixed decimal? etc

5. The VRF (Verifiable Random Function) section A.4 is empty, though this appears to already be tracked in polkadot-spec#467

6. Section 2.1 states

    > There is no assumption either on the size of the key nor on the size of the data stored under them, besides the fact that they are byte arrays with specific upper limits on their length. The limit is imposed by the encoding algorithms to store the key and the value in the storage trie.

    For clarity, consider explicitly specifying the associated length limits or link to a section that does.

Recommendations

The testing team recommends that these comments be discussed with relevant W3F and Parity team members responsible for the specification and reference implementation, and to consider implementing the suggestions above.

GSR-17	`VerifyBlockJustification()` Does Not Verify Justification Hash Relates to Block Being Justified		
Asset	`lib/grandpa/message_handler.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `VerifyBlockJustification(hash common.Hash, justification []byte)` verifies that a `justification` justifies a block with `hash`.

There is no check to ensure that `Justification.Hash` equals `hash`. As a result, it is possible that this justification is for a future block.

Substrate performs this check to ensure the justification is for this specific block and thus will reject justifications that are accepted by Gossamer.

The impact is rated low as the verification will only succeed if there are sufficient votes for a descendant of `hash` and thus a future block is already finalised.

Recommendations

The testing team recommends including a check that ensures the justification is for this specific block. That is, ensure `Justification.Hash` equals `hash` Alternatively update the specifications to allow justifications for descendants to be used.

GSR-18	Casting of `uint32` to `int` Will Overflow In 32-bit OS		
Asset	`lib/babe/verify.go`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `verifyPreRuntimeDigest()` verifies a digest message received from the network. It ensure the authority ID received in the message is in the current authority list by checking the ID is less than the length as seen in the snippet below.

```go
var authIdx uint32
switch d := babePreDigest.(type) {
case types.BabePrimaryPreDigest:
    authIdx = d.AuthorityIndex
case types.BabeSecondaryVRFPreDigest:
    authIdx = d.AuthorityIndex
case types.BabeSecondaryPlainPreDigest:
    authIdx = d.AuthorityIndex
}

if len(b.authorities) <= int(authIdx) {
    logger.Tracef("verifyPreRuntimeDigest invalid auth index %d, we have %d auths",
        authIdx, len(b.authorities))
    return nil, ErrInvalidBlockProducerIndex
}
```

An issue arises on 32-bit operating systems where `int` is a 32 bit signed integer. Casting a `uint32` which is larger than 2^{31} will result in a negative `int`. A negative number will not trigger the conditional statement above. The function will continue to execute and later use `authIdx` as an index to the `b.authorities` list which will panic due to an Index Out Of Bounds error.

Recommendations

Consider modifying the statement to include negative numbers, as seen below.

```go
if int(authIdx) < 0 || len(b.authorities) <= int(authIdx) {
    logger.Tracef("verifyPreRuntimeDigest invalid auth index %d, we have %d auths",
        authIdx, len(b.authorities))
    return nil, ErrInvalidBlockProducerIndex
}
```

GSR-19	Unrestricted Loop When Building Blocks		
Asset	`lib/babe/build.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is a loop when building BABE blocks that is unrestricted. While the slot has not finished, the builder will continue to poll the `transactionState` for new transactions. This is done in a loop without sleeps in between. The result is an excessive use of resources and constant grabbing and releasing of the transaction queue lock.

The following code block demonstrates the `for` loop which will execute constantly if the transaction queue is empty.

```
func (b *BlockBuilder) buildBlockExtrinsics(slot Slot, rt runtime.Instance) []*transaction.ValidTransaction {
  var included []*transaction.ValidTransaction

  for !hasSlotEnded(slot) {
    txn := b.transactionState.Pop()
    // Transaction queue is empty.
    if txn == nil {
      continue
    }
    ...
```

Recommendations

One solution is to add a small sleep when `txn == nil` to avoid continued, excessive polling of the `transactionState`.

GSR-20	Insecure Security Disclosure Policy		
Asset	`README.md`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Gossamer's current security policy is immature and recommends cleartext disclosure over an insecure medium of communication. No means for end-to-end encrypted communication is provided.

The advertised disclosure medium is as follows:

> Please email us a description of the flaw and any related information (e.g. reproduction steps, version) to security at chainsafe dot io. (Gossamer README.md)

Email is a non-private means of communication. While mail servers may exchange messages via TLS, there are no guarantees that each server a message passes through does so. Additionally, any mail server involved has full access to the message content.

The current severity is deemed low because Gossamer is not yet used in production. However, once in production where vulnerabilities could incur significant economic fallout, there is corresponding motivation for sophisticated actors to monitor or intercept sensitive communications (hence an increased risk).

Recommendations

Provide and publish a means for sensitive information to be securely communicated to select, trustworthy individuals who are responsible for Gossamer's security. This should involve end-to-end encryption. Advise that any sensitive security disclosures be made via this communication channel.

One solution could involve the same email address, but with the message contents encrypted via PGP/GPG.

If this is resolved via PGP/GPG, ensure the private key is appropriately stored in a secure manner.

The corresponding public key fingerprint should be published in several, independent places in order to minimse the risk that an attacker could replace the public key with their own. These could include the ChainSafe website, a public keyserver, `keybase.io`, Twitter and other social media platforms, ENS, DNS, and in the github organisation. Preferably, it should not be possible for an attacker to modify the fingerprint displayed in all these locations via a single compromised device.

Consider also allowing people to verify and sign this key during face-to-face meetings (or a master key).

GSR-21	Incorrect Use of Synchronisation Primitives		
Asset	`lib/trie/node.go` & `lib/trie/hash.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `encodingMu` reader/writer lock[6] appears intended to synchronise concurrent access to each `leaf` struct's `encoding` field. This is used incorrectly, potentially allowing for problematic race conditions.

Current functionality appears to not rely on the `encodingMu` for synchronisation. As such, this issue is deemed of *low* severity.

Detail

1. In `(*leaf).encodeAndHash()`, `l.encoding` is accessed outside of the bounds the lock at line [**267**] and line [**288**].

node.go
```go
263 func (l *leaf) encodeAndHash() (encoding, hash []byte, err error) {
      l.encodingMu.RLock()
265   if !l.isDirty() && l.encoding != nil && l.hash != nil {
        l.encodingMu.RUnlock()
267     return l.encoding, l.hash, nil
      }
269   l.encodingMu.RUnlock()

271   buffer := encodingBufferPool.Get().(*bytes.Buffer)
      buffer.Reset()
273   defer encodingBufferPool.Put(buffer)

275   err = encodeLeaf(l, buffer)
      if err != nil {
277     return nil, nil, err
      }
279
      bufferBytes := buffer.Bytes()
281
      l.encodingMu.Lock()
283   // TODO remove this copying since it defeats the purpose of `buffer`
      // and the sync.Pool.
285   l.encoding = make([]byte, len(bufferBytes))
      copy(l.encoding, bufferBytes)
287   l.encodingMu.Unlock()
      encoding = l.encoding // no need to copy
289   // ...
```

[6] A RWMutex, refer to `https://pkg.go.dev/sync#RWMutex`

2. Similarly in `hash.go` within `encodeNode()`, the value written to `n.encoding` at line [**117**] may no longer be accurate, because it relies on a value accessed in line [**106**] prior obtaining the current lock at line [**111**].

hash.go

```
105 case *leaf:
      err := encodeLeaf(n, buffer)
107   if err != nil {
        return fmt.Errorf("cannot encode leaf: %w", err)
109   }

111   n.encodingMu.Lock()
      defer n.encodingMu.Unlock()
113
      // TODO remove this copying since it defeats the purpose of `buffer`
115   // and the sync.Pool.
      n.encoding = make([]byte, buffer.Len())
117   copy(n.encoding, buffer.Bytes())
      return nil
```

Recommendations

Evaluate whether the `encodingMu` lock is required for correct functionality, as well as the `RWMutex` directly contained in the `leaf` and `branch` structs.

If these locking primitives are required, ensure the lock is held whenever `l.encoding` is accessed. Do not expect the value to remain unchanged when a lock is released and quickly re-acquired (e.g. when releasing a read lock to obtain the associated write lock).

When reasonable, perform regular unit and integration testing of concurrent components with the `-race` flag enabled.

GSR-22	BABE uses Milliseconds Rather Than Nanoseconds
Asset	`gossamer/lib/babe/babe.go`
Status	**Open**
Rating	Informational

Description

The function `getCurrentSlot()` calculates the current slot based off the clock time and the slot duration. The specification units for these times are Milliseconds. The Gossamer implementation uses Nanoseconds for these calculations as seen in the following code block.

```
func getCurrentSlot(slotDuration time.Duration) uint64 {
  return uint64(time.Now().UnixNano()) / uint64(slotDuration.Nanoseconds())
}
```

Recommendations

Consider updating the calculations to use Milliseconds rather than Nanoseconds.

GSR-23	Incorrect Log for `SealDigest` Verification
Asset	`gossamer/lib/babe/verify.go`
Status	Open
Rating	Informational

Description

There is an error in the logs when a `SealDigest` is not the last digest item. The output is currently *"first digest item is not pre-digest"* matching that of the `PreRuntimeDigest` error. This is not accurate as the error occurs when the last item is not a `SealDigest`.

```
// verifyAuthorshipRight verifies that the authority that produced a block was authorized to produce it.
func (b *verifier) verifyAuthorshipRight(header *types.Header) error {
  // header should have 2 digest items (possibly more in the future)
  // first item should be pre-digest, second should be seal
  if len(header.Digest.Types) < 2 {
    return fmt.Errorf("block header is missing digest items")
  }

  logger.Tracef("beginning BABE authorship right verification for block %s", header.Hash())

  // check for valid seal by verifying signature
  preDigestItem := header.Digest.Types[0]
  sealItem := header.Digest.Types[len(header.Digest.Types)-1]

  preDigest, ok := preDigestItem.Value().(types.PreRuntimeDigest)
  if !ok {
    return fmt.Errorf("first digest item is not pre-digest")
  }

  seal, ok := sealItem.Value().(types.SealDigest)
  if !ok {
    return fmt.Errorf("first digest item is not pre-digest")
  }
  // ...
```

Recommendations

Consider updating the log to something along the lines of *"last digest item is not seal digest"*.

GSR-24	Miscellaneous Gossamer Issues
Asset	`gossamer/*`
Status	**Open**
Rating	Informational

Description

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

1. **`lib/blocktree/blocktree.go` there is a code simplification:**
   `AddBlock()` has the following.

   ```
   number := big.NewInt(0)
   number.Add(parent.number, big.NewInt(1))
   ```

   This may be reduced to one line as `number := big.NewInt(0).Add(parent.number, big.NewInt(1))`.
2. **Use of "magic" numbers:** When the reason for their value is not obviously clear, it is generally preferable to replace literal number values with more descriptive constants. This improves maintainability via a clearer reading experience, helps minimise unnoticed bugs caused by typos, and makes refactoring easier should these values later change. For example, literal `1`, `2`, and `3` values are used throughout `lib/trie/node.go` to refer to the type of the node, as encoded in the header. It is not obvious to a casual reader what these values mean and would be better described as something like `LEAF`, `BRANCH_NO_VALUE`, `BRANCH_WITH_VALUE` respectively.
3. **`lib/grandpa/vote_message.go` has a comment is in the wrong spot.**
   On line [**196**] there is the comment
   `// check for equivocation ie. multiple votes within one subround`. It should be around line [**230**] when we check the equivocation.
4. **Fragile links used in GoDoc comments:**
   Comments often link to the Substrate reference implementation. However, when referring to specific lines of the code, the link should be pinned to a specific commit or tag in order to ensure they resolve meaningfully. The following comments were identified referring to specific line numbers within the non-specific master branch.
   - `lib/babe/crypto.go:31`
   - `lib/crypto/keypair.go:59`
   - `lib/common/hasher_test.go:49`
   - `lib/grandpa/message.go:52`
   - `lib/babe/crypto.go:31,85`
   - `lib/babe/secondary.go:30,48`
5. **Typos and Nitpicks:**
   - In the comment at `lib/trie/codec.go:48`, "nibblesToKey" should be "nibblesToKeyLE".

Recommendations

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

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided to the development team alongside this document. Output is as follows:

```
$ go test -v ./tests/...
=== RUN   TestEmptyTrieHash
    trie_test.go:12: Val: 0x03170a2e7597b7b7e3d84c05391d139a62b157e78786d8c082f29dcf4c111314
--- PASS: TestEmptyTrieHash (0.00s)
PASS
```

```
$ cd diff_fuzzing && cargo test
running 9 tests
test util::tests::make_authority_index ... ok
test util::tests::check_generate_auths_zero ... ok
test util::tests::test_from_split_one ... ok
test util::tests::check_dummy_auth ... ok
test util::tests::test_from_split_order ... ok
test util::tests::test_from_split_zero ... ok
test util::tests::test_from_split_simple ... ok
test util::tests::test_from_split_max ... ok
test common::tests::test_diff_trie_root_crash ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running tests/test_rust_trie.rs (target/debug/deps/test_rust_trie-d90bb4241358efdf)

running 4 tests
test test_are_empty_and_child_the_same ... ok
test test_trie_root_empty ... ok
test test_trie_root_ordering ... ok
test test_trie_root_empty_key ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

```
$ cd diff_fuzzing/gossamer_adapters && cargo test
running 11 tests
test bindgen_test_layout_BytesPair ... ok
test bindgen_test_layout_GoInterface ... ok
test bindgen_test_layout_GoSlice ... ok
test bindgen_test_layout_GossamerFuzzerTestBabeThreshold_return ... ok
test bindgen_test_layout__GoString_ ... ok
test bindgen_test_layout___fsid_t ... ok
test bindgen_test_layout_max_align_t ... ok
test tests::basic_conversion ... ok
test tests::it_works ... ok
test tests::basic_babe_threshold ... ok
test tests::empty_trie_root ... ok

test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Appendix B 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.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

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