Protocol Labs

Lotus Security Assessment

Version: 3.1

October, 2020

Contents

Introduction

Protocol Labs is a research, development, and deployment institution for improving Internet technology. Protocol Labs leads groundbreaking internet projects, such as *IPFS*, a decentralized web protocol; and *libp2p*, a modular network stack for peer-to-peer applications.

Filecoin is an open source project led by Protocol Labs which aims at providing a decentralized storage network that turns cloud storage into an algorithmic market. Miners earn the native protocol token by providing data storage and/or retrieval.

Lotus is the reference implementation of the Filecoin distributed storage network, developed in Golang.

Sigma Prime was approached by Protocol Labs to perform a time-boxed security assessment of Lotus, along with a selected set of its external dependencies.

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 contract. 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 Lotus client 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 found within the code.

Overview

Lotus is an implementation of the Filecoin Distributed Storage Network, written in Golang and is composed of two separate binaries:

- **The Lotus Daemon:** Core component responsible for handling the Blockchain node logic by handling peer-to-peer networking, chain syncing, block validation, data retrieval and transfer, etc.
- **The Lotus Storage Miner:** Mining component used to manage a single storage miner by contributing to the network through *Sector* commitments and *Proofs-of-Spacetime* data proving it is storing the sectors it has committed to. This component communicates with the Lotus daemon via JSON-RPC API calls.

This security review targeted the following scope items:

- **Part A: Core Logic:** `filecoin-project/lotus` :
  - Main repository containing the Golang reference implementation of the Filecoin protocol. This codebase incorporates the core logic of a Filecoin node, including the state tree model, the chain syncing protocol, the core storage power consensus mechanism (SPC), etc.
- **Part B: Specs Actors:** `filecoin-project/specs-actors` :
  - Repository defining the business logic and permissions for the different types of actors in the Filecoin network (account actors, storage market actors, payment channel actors, reward actors, etc.).
- **Part C: Go Fil Markets:** `filecoin-project/go-fil-markets` :
  - Modular implementations of the storage and retrieval market subsystems of Filecoin (filestore, pieceio, piecestore, storage market, retrieval market).
- **Part D: Selected Lotus Dependencies:**
  - filecoin-project/go-address
  - filecoin-project/go-amt-ipld
  - filecoin-project/go-bitfield
  - filecoin-project/go-cbor-util
  - filecoin-project/go-crypto
  - filecoin-project/go-data-transfer
  - filecoin-project/go-fil-commcid
  - filecoin-project/go-padreader
  - filecoin-project/go-sectorbuilder
  - filecoin-project/go-statemachine
  - filecoin-project/go-statestore
  - filecoin-project/sector-storage
  - filecoin-project/specs-storage
  - filecoin-project/storage-fsm
  - ipfs/go-graphsync
  - ipfs/go-hamt-ipld
  - ipfs/go-ipld-cbor
  - whyrusleeping/cbor-gen

Security Review Summary

This review was initially conducted on the following commits:

- `filecoin-project/lotus` : 7d166fd7
- `filecoin-project/specs-actors` : 0df536f
- `filecoin-project/go-fil-markets` : 19255be

With dependencies at the following commits:

- `whyrusleeping/cbor-gen` : 6496743
- `filecoin-project/go-address` : 8b6f2fb
- `filecoin-project/go-amt-ipld` : 6263827
- `filecoin-project/go-bitfield` : 4d77652 (Release v0.0.1)
- `filecoin-project/go-cbor-util` : 08c40a1
- `filecoin-project/go-crypto` : effae4e
- `filecoin-project/go-data-transfer` : a74d66b (Release v0.3.0)
- `filecoin-project/go-fil-commcid` : 2b8bd03
- `ipfs/go-graphsync` : 9d5f2c2
- `ipfs/go-hamt-ipld` : d53d20a
- `ipfs/go-ipld-cbor` : a0bd04d
- `filecoin-project/go-padreader` : 5482570
- `filecoin-project/go-statemachine` : 2074af6
- `filecoin-project/go-statestore` : 2ee326d (Release v0.1.0)
- `filecoin-project/sector-storage` : a59ca75
- `filecoin-project/specs-storage` : 61b2d91
- `filecoin-project/storage-fsm` : 83fd743

Fuzzing activities leveraging go-fuzz and libFuzzer have been performed by the testing team in order to identify panics within the scope of this assessment. `go-fuzz` is a coverage-guided tool which explores different code paths by mutating input to reach as many code paths as possible. The aim is to find memory leaks, overflows, index out of bounds or any other panics.

Specifically, the testing team produced a total of *one hundred and fifty-eight* (158) fuzzing targets. These fuzzing targets have all been shared with the development as a byproduct of this security review. Execution and instrumentation can be done using a `Makefile`, by simply running `make run-fuzz-${TARGET-NAME}` (targets list and detailed instructions are available inside the `Makefile`).

Along with fuzzing activities, the testing team performed tailored and targeted testing to identify some of the vulnerabilities raised in this report. The relevant test scripts and test vectors have also been shared with the Lotus team.

A second roud of testing was performed by Sigma Prime, which lead to the identification of seven (7) additional vulnerabilities.

The testing team identified a total of thirty-three (33) issues during the first round of this assessment (`LOT-01` to `LOT-33`), of which:

- Nine (9) are classified as critical risk,
- One (1) is classified as high risk,
- Three (3) are classified as medium risk,
- Eleven (11) are classified as low risk,
- Nine (9) are classified as informational.

The testing team identified a total of seven (7) issues during the second round of this assessment (`LOT-34` to `LOT-40`), of which:

- One (1) is classified as critical risk,
- One (1) is classified as medium risk,
- Five (5) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the scope of this assessment. 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 code base, including comments not directly related to the security posture of Filecoin Proving Subsystem, 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
LOT-01	Ineffective `BlockHeader` Equality Check	Critical	Resolved
LOT-02	`nil` Pointer Accesses from `FromNet()`	Critical	Resolved
LOT-03	Panic from Slice Indexing in Graphsync	Critical	Resolved
LOT-04	Insufficient Validation in `NewTipSet()` Resulting in DoS	Critical	Resolved
LOT-05	Insufficient Validation of `BlockSyncResponse` Message Resulting in DoS	Critical	Resolved
LOT-06	Panic in `UnmarshalCBOR` for `Deferred` type	Critical	Resolved
LOT-07	`BlockSync` Client Not Verifying Tipset Received was Requested	Critical	Resolved
LOT-08	`(*BlockSync).GetBlocks()` Fails to Query Peers After Receiving Malformed Response	Critical	Resolved
LOT-09	Insufficient Validation of `store.FullTipSet` Exposes DoS Vulnerability	Critical	Resolved
LOT-10	`TipSet` Height is Not Validated As Part of a Chain	High	Resolved
LOT-11	Unlimited Active Peer Connections Allows DoS	Medium	Closed
LOT-12	Withdraw Other Users Balance from the Market Actor	Medium	Resolved
LOT-13	Swapping a Signer in `MultiSigActor` Allows for a Signer to Approve a Transaction Twice	Medium	Resolved
LOT-14	`StorageMarket` `ConnManager` Unbound Number of Streams	Low	Resolved
LOT-15	Deterministic Peer Ordering Can Be Abused	Low	Closed
LOT-16	`SettleDelay` is too Low in `PaychActor`	Low	Resolved
LOT-17	Early Deal Termination from Sector Expiry	Low	Closed
LOT-18	Sector Expiration not Validated in `ExtendSectorExpiration()`	Low	Closed
LOT-19	Generated `UnmarshalCBOR` Methods Allow Omission of Struct Fields	Low	Closed
LOT-20	No Bounds Checks for `(*Int).MarshalCBOR()`	Low	Resolved
LOT-21	Validation of Block Message Roots Persists Data to Storage	Low	Closed
LOT-22	`ReadByteArray()` Not Using `maxlen` Argument to Reject Oversized Buffers	Low	Resolved

LOT-23	Potential to Reduce `ToSend` in `PaychActor`	Low	Closed
LOT-24	API Call to `ChainStatObj` Crashes Daemon	Low	Resolved
LOT-25	Transfers to Certain Actors Results in Lost Funds	Informational	Open
LOT-26	Validation of Piece Sizes	Informational	Open
LOT-27	`VerifiedRegistryRoot` Address is Constant	Informational	Open
LOT-28	`MultiSigActor.Constructor()` Parameter Verification	Informational	Resolved
LOT-29	`MultiSigActor` Can Manipulate `NumApprovalsThreshold` to an Invalid State	Informational	Resolved
LOT-30	Potential `nil` Pointer Access in `ReportConsensusFault()`	Informational	Closed
LOT-31	`BlockSync.processBlocksResponse` Does Not Verify Length	Informational	Resolved
LOT-32	Miscellaneous Observations in the Lotus & Specs-Actors Codebases	Informational	Resolved
LOT-33	Miscellaneous Observations in Dependencies	Informational	Resolved
LOT-34	Unmarshalling Signatures may Panic	Critical	Resolved
LOT-35	Storage Miner Custom Deal Filter Logic	Medium	Resolved
LOT-36	Storage Market `DealProposals` Validation	Informational	Open
LOT-37	Wallet Keystore Secret Information is Stored Unencrypted	Informational	Open
LOT-38	JSON Web Token is Stored Unencrypted	Informational	Open
LOT-39	Retrieval Path is Local to Daemon not Caller	Informational	Open
LOT-40	Client Imports Small Piece Sizes	Informational	Open

LOT-01	Ineffective `BlockHeader` Equality Check		
Asset	`lotus/chain/vm/syscalls.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `VerifyConsensusFault()` (as part of the virtual machine) takes two `BlockHeader`s as input and determines if they meet the criteria for the producer to be slashed (who must be the same for both blocks).

In order to meet the slashing conditions the two blocks must be different. To check the blocks are different the following comparison is made: `if bytes.Equal(a, b) { return ... }`, where `a` and `b` are blocks represented in bytes.

However, `BlockHeader.UnmarshalCBOR()` is not an injective function. Thus, it is possible to have two different byte representations which unmarshal to the same block. More specifically, `BlockHeader.UnmarshalCBOR()` allows for bytes to be appended, which are ignored by the unmarshalling function.

A simple example of the "injectiveness" exploit would be to take a valid signed `BlockHeader` and append a single zero byte to the end i.e. call `VerifyConsensusFault(a, b, extra []byte)` with `a` set to the valid `BlockHeader` bytes and `b = a + 0`. This would then pass the `bytes.Equal(a, b)` check, unmarshal the headers and begin to check if the two blocks meet any of the slashing criteria.

This attack would raise the **"double-fork mining fault"**, which checks that `blockA.Height == blockB.Height`. This will succeed, as `blockA` and `blockB` are identical blocks.

The miner who produced the valid block will now be slashed for double-fork mining when they have only produced a single block at this height.

Recommendations

Although `blockA.DeepEquals(blockB)` may protect against the attack, any injectiveness bugs in `MarshalCBOR()` will result in a similar exploit.

We recommend performing the equality check between blocks by calling `blockA.SigningBytes()` and `blockB.SigningBytes()` and checking whether the output of these are equal.

Resolution

This issue has been resolved by performing the block equality check over the block `Cid`. A block `Cid` is unique for each block, thus ensuring any two blocks with the same `Cid` are the same block.

The issue has been resolved in commit e1c9c42.

LOT-02	`nil` Pointer Accesses from `FromNet()`		
Asset	`go-data-transfer/message/message.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`DataTransferMessage`s are deserialised by the function `FromNet()`, which takes raw bytes passed from the `libp2pDataTransferNetwork` function `handleNewStream()`.

`FromNet()` will return `(DataTransferMessage, error)`, with the error depending on the successful deserialisation of the bytes.

As seen in line [87] of the snippet below, in the case where the `transferMessage` is a request the returned error is always `nil`.

```
func FromNet(r io.Reader) (DataTransferMessage, error) {
84   tresp := transferMessage{}
     err := tresp.UnmarshalCBOR(r)
86   if tresp.IsRequest() {
       return tresp.Request, nil
88   }
     return tresp.Response, err
90 }
```

message.go

The implications are that an unsuccessful deserialisation will result in `tresp.Request = nil` and `error = nil`. In the libp2p implementation when the error is `nil`, it will call `graphsyncReceiver.ReceiveRequest()`, which calls `incoming.VoucherType()`, causing a `nil` pointer panic and thereby crashing the node.

Passing the bytes `0x83f5` as a `DataTransferMessage` is sufficient to reach the case where `FromNet()` returns `nil, nil`.

A similar issue exists due to `transferMessage.UnmarshalCBOR()`. The `transferMessage` struct (as seen below) unmarshals three elements of the struct. Here both `Request` and `Response` can be set to `nil` simultaneously, and `transferMessage.UnmarshalCBOR()` will return `error = nil`.

```
10 type transferMessage struct {
     IsRq bool
12
     Request  *transferRequest
14   Response *transferResponse
   }
```

transfer_message.go

The implications of this are, again, that `FromNet()` returns `nil, nil`.

If `IsRq == true` it will take the same code path as described above and cause the same panic.

If `IsRq == false` a `nil` pointer panic will occur when `graphsyncReceiver.ReceiveResponse()` calls `incoming.Accepted()`.

Passing bytes `0x83f5f6f6` and `0x83f4f6f6` respectively will cause the two cases.

Recommendations

To solve the first issue it is recommended to propagate the error in `FromNet()` for the case where `IsRequest() == true`.

The recommendation for the second issue is to enforce that exactly one of `Request` or `Response` is `nil` in `(*transferMessage).UnmarshalCBOR()`.

Resolution

The error case is now handled for `FromNet()` when the `transferMessage` is a request. The issue was resolved in PR #59. Additionally, the cases where `FromNet()` returns `(nil, nil)` has been updated to return an error in PR #74.

LOT-03	Panic from Slice Indexing in Graphsync		
Asset	`go-graphsync/message/pb/message.pb.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Messages received by `libp2pGraphSyncNetwork` call `FromPBReader()`, which attempts to deserialise the bytes received from the network.

The bytes are deserialised in the function `FromPBReader()`. Due to an overflow, the function may try to access an index out of bounds, causing a panic and the Lotus node to crash.

As seen in the excerpt below on line [1012], if `postStringIndexmapkey := iNdEx + intStringLenmapkey` overflows, it may cause the `dAtA` array to be accessed at a large negative index, causing an "index out of bounds" panic.

```
1012 postStringIndexmapkey := iNdEx + intStringLenmapkey
     if postStringIndexmapkey > l {
1014   return io.ErrUnexpectedEOF
     }
1016 mapkey = string(dAtA[iNdEx:postStringIndexmapkey])
```

message.pb.go

Recommendations

A check should be added to ensure that no overflow has occurred. This can be done by ensuring `postStringIndexmapkey >= iNdEx`.

Resolution

This issue has been resolved by checking for negative indexing in the PR #79.

LOT-04	Insufficient Validation in `NewTipSet()` Resulting in DoS		
Asset	`lotus/chain/types/tipset.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`NewTipSet()` fails to verify that each `BlockHeader` (passed as input) contains the same number of *"Parents"*, exposing the Lotus node to a crash or incorrectly accepting an invalid tipset.

The relevant section of `NewTipSet()` is listed below.

```
  for i, cid := range b.Parents {
116   if cid != blks[0].Parents[i] {
        return nil, fmt.Errorf("cannot create tipset with mismatching parents")
118   }
  }
```

tipset.go

After sorting with `tipsetSortFunc`, each subsequent block header has the contents of its `Parents` field compared to the first entry (`blks[0]`), but the lengths of the `Parents` fields are not initially verified to be equal. If we consider some entry b (`b != blks[0]` where `len(b.Parents) != len(blks[0].Parents)`), there are two interesting cases:

i) `len(b.Parents) > len(blks[0].Parents)` — Here, an *"index out of range"* panic is triggered at line [106] once `i >= len(blks[0].Parents)`.

ii) `len(b.Parents) < len(blks[0].Parents)` — If `b` contains a suitable subset of `blks[0].Parents` such that `b.Parents == blks[0].Parents[:len(b.Parents)]`, it is possible for the mismatch check (listed above) to pass even though the blocks have differing parents.

Because `NewTipSet()` receives untrusted input (e.g. when used by `(*TipSet).UnmarshalCBOR()` or in `(*Syncer).ValidateTipSet` at `sync.go:463`), this can be exploited by an external attacker to perform a Denial-of-Service (DoS) attack and is deemed a critical risk.

The testing team was unable to identify any subsequent validation steps that check this property (including `ValidateTipset` at `lotus/chain/sync.go:457`). As such, it appears possible — via case ii) — for the Lotus node to accept a tipset with inconsistent parents as part of its heaviest chain. If another Filecoin implementation were to correctly invalidate such a tipset, the network could fork, with Lotus building on the invalid chain and other clients building on a separate one.

Recommendations

In `NewTipSet()`, check that each `BlockHeader` has the same number of parents before iterating through them.

Ensure these checks are performed whenever a `TipSet` is constructed from untrusted input.

Resolution

The tipset validation has been updated such that it first checks the length of each of the `parents` arrays before indexing them, in commit 21ffe18.

LOT-05	Insufficient Validation of `BlockSyncResponse` Message Resulting in DoS		
Asset	`lotus/chain/blocksync/blocksync.go` & `lotus/chain/sync.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

During syncing or the initial *"Hello"* exchange, a malicious peer can send a malformed `BlockSyncResponse` message to trigger a crash on the Lotus node. This vulnerability is a result of insufficient validation of the `BSTipSet` type, allowing an"index out of range" panic when converting it to a `types.TipSet` in `zipTipSetAndMessages()` (`sync.go:307-322`) and `bstsToFullTipSet()` (`blocksync.go:246-251`).

The `BlsMsgIncludes` and `SecpkMsgIncludes` `BSTipSet` fields contain indices into the `BlsMessages` and `SecpkMessages` lists respectively, intended to associate each BlockHeaders with their messages while avoiding storage of duplicate messages.

However, there is no prior validation to ensure that every block has a corresponding "Includes" entry (i.e. that `len(bts.Blocks) == len(bts.BlsMsgIncludes) && len(bts.Blocks) == len(bts.SecpkMsgIncludes)`). Similarly, no validation ensures each `XXXMsgIncludes[i][j]` value is an appropriate index into the relevant `XXXMessages` list. As such, "index out of range" panics can occur at each of lines [246,247,249,250].

```
240 func bstsToFullTipSet(bts *BSTipSet) (*store.FullTipSet, error) {
      fts := &store.FullTipSet{}
242   for i, b := range bts.Blocks {
        fb := &types.FullBlock{
244       Header: b,
        }
246     for _, mi := range bts.BlsMsgIncludes[i] {
          fb.BlsMessages = append(fb.BlsMessages, bts.BlsMessages[mi])
248     }
        for _, mi := range bts.SecpkMsgIncludes[i] {
250       fb.SecpkMessages = append(fb.SecpkMessages, bts.SecpkMessages[mi])
        }
252
        fts.Blocks = append(fts.Blocks, fb)
254   }

256   return fts, nil
    }
```

blocksync.go

Although `zipTipSetAndMessages()` appropriately checks the lengths of the lists (avoiding crashes equivalent to `blocksync.go:246,249` , see below), it is similarly missing validation that the `bmi[i][j]` and `smi[i][j]` values are appropriate indices for the `allbmsgs` and `allsmsgs` lists.

```
302   if len(ts.Blocks()) != len(smi) || len(ts.Blocks()) != len(bmi) {
        return nil, fmt.Errorf("msgincl length didnt match tipset size")
304   }
```

sync.go

Recommendations

Perform sufficient bounds checking of the `BlsMsgIncludes` and `SecpkMsgIncludes` indices when processing untrusted `BSTipSet` values.

Additionally, consider checking that the `XXXMsgIncludes` lists contain no duplicate index values, which indicates that a block contains the same message twice.

Add unit tests for `bstsToFullTipSet()` and `zipTipSetAndMessages()` that include passing invalid `BSTipSet` arguments and checking that the function does not panic.

Resolution

This was by PRs #2715 and #3939.

All untrusted `Response` objects (renamed from `BlockSyncResponse`) are converted to `validatedResponse` structs by `(*BlockSync).processResponse()`. When the `Response` contains both messages and block headers (i.e. the request was made with both `IncludeHeaders` and `IncludeMessages` options), `(*BlockSync).processRespons` effectively validates that all indices in the contained `XXXIncludes` fields are in a valid range. This occurs during calls to `(*BlockSync).GetFullTipSet()`.

When the response contains only messages (which occurs during calls to `(*BlockSync).GetChainMessages()`), the tipset's block headers must be provided separately in order to validate the `CompactedMessages`' block indices.

This has indeed been done: In PR #3939, `(*BlockSync).GetChainMessages()` now takes as a parameter all block headers for which should collect messages (in the form of `TipSet` structs).

LOT-06	Panic in `UnmarshalCBOR` for `Deferred` type		
Asset	`whyrusleeping/cbor-gen/utils.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

When calling `UnmarshalCBOR` on a `Deferred` type, malformed bytes cause a panic. Due to the use of *CBOR* in communication amongst nodes and clients, malicious actors can send bytes throughout the network which will trigger panics upon reception.

The `UnmarshalCBOR` for `Deferred` types can be observed throughout the codebase where network messages are received and processed. Given malformed bytes, the panics produced are:

- `panic: runtime error: makeslice: len out of range`
- `runtime: out of memory: cannot allocate 140944456286208-byte block (66813952 in use)`
  `fatal error: out of memory`

This bug originates from the `whyrusleeping/cbor-gen` dependency.

```
func (d *Deferred) UnmarshalCBOR(br io.Reader) error {
    // TODO: theres a more efficient way to implement this method, but for now
    // this is fine
    maj, extra, err := CborReadHeader(br)
    if err != nil {
        return err
    }
    header := CborEncodeMajorType(maj, extra)

    switch maj {
        case MajUnsignedInt, MajNegativeInt, MajOther:
            d.Raw = header
            return nil
        case MajByteString, MajTextString:
            buf := make([]byte, int(extra)+len(header))
            copy(buf, header)
            if _, err := io.ReadFull(br, buf[len(header):]); err != nil {
                return err
            }

            d.Raw = buf

            return nil
        ...
```

whyrusleeping/cbor-gen/utils.go

The issue can be observed in a number of objects throughout the codebase. However, the following structures are exposed to the network and can be abused by malicious actors to cause panics on nodes by sending malformed bytes.

We defer the fuzzing output and further explanation to the Appendix A

- **BlockMsg** used in `lotus/chain/types/blockmsg.go:17` to read the CBOR of the RPC for any block messages.
- **BlockSyncResponse** used in `lotus/chain/blocksync/blocksync.go:97` to read the CBOR RPC in the `HandleStream` for incoming RPC reads.
- **DealProposal** used in `go-fil-markets/retrievalmarket/network/deal_stream.go:24` in function `ReadDealProposal` to read deal proposals from the network when negotiating retrieval deals.
- **SignedMessage** used in `lotus/chain/types/signedmessage.go:51` in function `DecodeSignedMessage` to read signed messages from the network.
- **TipSet** used in `lotus/chain/types/tipset.go:72` in function `(*TipSet).UnmarshalCBOR` to decode the TipSet.
- **SignedVoucher** used in `filecoin-project/specs-actors/actors/builtin/paych/cbor_gen.go:612` which is used when negotiating and performing retrieval market deals in `go-fil-markets/retrievalmarket/impl/providerstates/provider_states.go::ProcessPayment`.

There are also a number of objects in the dependencies that can produce this panic:

- **filecoin-project/go-amt-ipld/cbor_gen.go:189** in function `(*Node).UnmarshalCBOR(br io.Reader)`.
- **ipfs/go-hamt-ipld/pointer_cbor.go:116** in function `(*Pointer).UnmarshalCBOR(br io.Reader)`.
- **filecoin-project/go-data-transfer/message/message.go:85** in function `FromNet(r io.Reader)`, which calls:
  - `transfer_message_cbor_gen.go:88`
  - `transfer_request_cbor_gen.go:188`

Recommendations

The length of the bytes should be checked before creating the slice in the `UnmarshalCBOR` of `whyrusleeping/unmarshal-cbor`.

This should be applied to the upstream library to fix all dependent structures amongst this codebase.

Resolution

The issue was fixed upstream in `whyrusleeping/cbor-gen` in commit 4ef2d6a by checking indices and enforcing max lengths when allocating arrays.

LOT-07	`BlockSync` Client Not Verifying Tipset Received was Requested		
Asset	`lotus/chain/blocksync/blocksync.go` & `lotus/chain/blocksync/blocksync_client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

When processing `TipSet`s requested from a peer in `(*BlockSync).GetFullTipSet` and `(*BlockSync).processBlocksResponse` (used in `GetBlocks()`), the `BlockSync` client does not verify that a received `TipSet` is what was actually requested (or, for a chain, that the head of the chain matches the request.)

In certain circumstances, a malicious actor may be able to use this to trick a Lotus node into unknowingly marking valid blocks as *Bad*, causing the node to fork from the network and likely receive punishment for not responding to relevant challenges.

In detail, given a `BlockSyncResponse resp` responding to some `BlockSyncRequest req`, there is no consistent check that `resp.Chain[0]` matches `req.Start`.

In `lotus/chain/blocksync/blocksync_client.go`, the function call `processBlocksResponse` does not ensure that the requested `TipSet` is actually the start of the `res.Chain[0].Blocks`, so the retrieved blocks could be a forked chain of blocks.

Although `(*Syncer).CollectChain()` checks that the start of the chain matches the requested `TipSet` (listed below), when the chain is composed from multiple calls to `(*BlockSync).GetBlocks()` only the first call is appropriately verified to return a chain starting with the requested `TipSet`. As such, subsequent responses may be entirely disjoint from the initial chain.

```
1298  if !headers[0].Equals(ts) {
        log.Errorf("collectChain headers[0] should be equal to sync target. Its not: %s != %s",
        headers[0].Cids(), ts.Cids())
      }
```

sync.go

These disjoint chains may be properly validated (or ignored) during later state transitions but serious impact can occur within the `Syncer`. Upon receiving a second (or later) `(*BlockSync).GetBlocks()` call from in `(*Syncer).collectHeaders()`, a malicious actor can respond with a disjoint chain containing a known *Bad* block (in the node's `badBlockCache`). Because this is incorrectly thought to be part of the same chain, valid blocks from the first response can be incorrectly marked as *Bad*, even though those bad blocks are not actually ancestors.

This is not checked during `(*BlockSync).GetFullTipSet()`, so the `TipSet` contained in the response may not match the `HelloMessage`. Note that the testing team have not identified any associated exploits at this stage (the peer would just be corrupting data associated with itself).

A malicious actor may use this in combination with LOT-10 to more easily cause `(*Syncer).collectHeaders()` to loop indefinitely. Because of this, the chain segments can be disjoint, allowing the actor to reuse existing valid

or self-created chain segments (as long as the height remains above `untilHeight`), rather than needing to construct a chain of indefinite length. As in LOT-10, the actor can repeat this attack `syncWorkerCount = 3` times, to stall all syncing for the Lotus node.

Recommendations

Validate that the chain received in every `BlockSyncResponse` correctly starts with the corresponding `BlockSyncRequest.Start` TipSet. We recommend verifying this early in the validation process (e.g. in `(*BlockSync).processBlocksResponse` and `(*BlockSync).GetFullTipSet()`).

Ensure any peers providing non-matching responses are appropriately evicted and blacklisted for malicious behavior, or at least demote it as a failure response.

Consider implementing unit tests that provide invalid and malicious data to `processBlocksResponse()` (and similar functions), to identify and avoid regressions.

Resolution

All `Response`[1] objects are passed to `(*BlockSync).processResponse()`, which checks that the response's first block header matches the request (at `client.go:190`). This was resolved as part of PR #2715.

[1] The `BlockSyncResponse` type has since been renamed to `Response`

LOT-08	`(*BlockSync).GetBlocks()` Fails to Query Peers After Receiving Malformed Response		
Asset	`lotus/chain/blocksync/blocksync_client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In `(*BlockSync).GetBlocks()` at line [189], `(*BlockSync).processBlockResponse()` returns an error when unable to successfully process a `Success` response received from a peer. Instead of querying subsequent peers, `GetBlocks()` immediately returns, propagating the error. This error response halts sync progress in the calling methods `(*Syncer).collectHeaders()` and `(*Syncer).syncFork()`.

Because the `BlockSyncResponse` is untrusted input from the network (with the only prior validation being that it unmarshals without error), a malicious peer can readily send a response containing a `Success` status and a malformed chain.

Furthermore, if a malicious actor can reliably control the top 5 active peers, they can perform a denial of service attack and prevent syncing from completing. This is because `shufflePrefix()` only permutes the top 5 peers, so the malicious actor will have guaranteed control of the peer first queried by `GetBlocks()`, and can prevent any querying of subsequent peers. Perhaps worse is a possibility that the attacker can use this eclipse-like attack to selectively allow syncing of only desired chains, potentially causing slashing or enabling a double-spend.

Because peers are deterministically ordered based on latency and error rates[2], a targeted, sophisticated attacker can reasonably and reliably control the top 5 peers. Because no failure is logged when failing to process the malformed `Success` response, the malicious peer will not be demoted in the peer ranking, and can repeat the attack without issue.

Recommendations

In `(*BlockSync).GetBlocks()`, continue to query other peers when `processBlockResponse()` returns an error value.

Consider evicting and/or blacklisting peers that respond with malformed chains. This recommendation follows the notion that there is no legitimate reason for friendly peers to provide malformed responses with a `Success` status. Although an attacker could generate fresh peers, the cost of any attack should increase.

Resolution

`(*BlockSync).GetBlocks()` now delegates to `(*BlockSync).doRequest()`, which continues to query subsequent peers whenever an invalid response is returned (according to `(*BlockSync).processResponse()`). This was resolved as part of PR #2715.

[2] See `(*bsPeerTracker).prefSortedPeers()` and LOT-15

LOT-09	Insufficient Validation of `store.FullTipSet` Exposes DoS Vulnerability		
Asset	`lotus/chain/store/fts.go` , `lotus/chain/sync.go` & `lotus/chain/blocksync/blocksync.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

As part of processing a new peer's `(lotus/node/hello).HelloMessage` and syncing, Lotus nodes construct `store.FullTipSet` s from untrusted `blocksync.BSTipSet` structures.[3] The `store.FullTipSet` is constructed in `bstsToFullTipSet()` [4] and `zipTipSetAndMessages()` [5] respectively.

Although some processing is done to construct the `FullTipSet` , more validation occurs during the creation of its contained `TipSet` in `(*FullTipSet).TipSet()` (with the call to `types.NewTipSet()`). Because `types.NewTipSet()` checks properties that have not been previously validated (like consistent block parents and height), it can reasonably fail. However, the current implementation of `(*FullTipSet).TipSet()` assumes equivalent validation has already occurred and `NewTipSet()` *must* succeed. A panic is emitted at `fts.go:47` if this is not the case.

This panic occurs when `(*FullTipSet).TipSet()` is first called:

- For syncing, this panic is emitted from `(*Syncer).ValidateTipSet()` at `sync.go:461` (or `sync.go:1171` `syncMessagesAndCheckState()` if log arguments are always evaluated).
- When processing a `HelloMessage` , this is emitted from `hello.go:111` `if ts.TipSet().Height() > 0` .

As such, any malicious peer can send `BlockSyncResponse` messages containing blocks with inconsistent parents or heights and cause the Lotus node to crash. Because the `BlockSyncResponse` is a *response*, a malicious peer must receive a `BlockSyncRequest` message from its target in order to exploit this vulnerability.

When the target is syncing, the malicious peer would need to be high in the peer ordering to have a reasonable chance of receiving the request. Because this requires low latency to the peer, the attacker would likely need to target specific nodes.

However, *any* new peer connection will perform a "hello" handshake and send a `BlockSyncRequest` . Thus, a single malicious peer can reliably crash nodes en masse.

[3] Contained in `blocksync.BlockSyncResponse` messages.
[4] Located at `blocksync.go:240` .
[5] Located at `sync.go:301` .

Recommendations

Ensure any construction of `FullTipSet` checks *all* properties validated by `NewTipSet()`, or otherwise allow `(*FullTipSet).TipSet()` to return an error.

Implement unit tests to check this and avoid regressions.

Consider centrally implementing a constructor for `FullTipSet` (similar to `types.NewTipSet()`) that checks these properties, and require that it be used to construct any `FullTipSet` in the codebase. It may be preferable for this to directly call `types.NewTipSet()` and populate the `FullTipSet.tipset` field.

Resolution

Any `FullTipSet` objects created in blocksync are now constructed in `(*validatedResponse).toFullTipSets()`. `validatedResponse` objects are only created in `(*BlockSync).processResponse()`, which makes use of `types.NewTipSet()` to ensure all relevant checks are performed. Although `zipTipSetAndMessages()` still directly constructs `FullTipSet`, the `TipSet`s passed as arguments originate from `(*Syncer).collectHeaders()` and thus `(*BlockSync).GetBlocks()` (which also passes through `(*BlockSync).processResponse()`).

This was resolved as part of PR #2715.

LOT-10	`TipSet` Height is Not Validated As Part of a Chain		
Asset	`lotus/chain/blocksync.go` & `lotus/chain/stmgr/stmgr.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When receiving `TipSet`s from the network, validation in `types.NewTipSet()` ensures that each block in the `TipSet` has a consistent height. Similarly, `(*BlockSync).processBlocksResponse()` and `(*StateManager).ValidateChain` check that the chain is correctly joined via the `Parents` field.

However, no code appears to verify that the `TipSet`s' height values are strictly decreasing. Several sections appear to rely on this value, even though a malicious actor can currently send a side-chain or propose a block containing an unexpected height value. Reliant code includes:

- In `(*Syncer).collectHeaders` at `sync.go:1035,1081`, with the loop condition at line [1035]: `for blockSet[len(blockSet)-1].Height() > untilHeight {` a malicious actor can construct an invalid chain containing TipSets of an unchanging height set to some value much larger than the current height. This thread can then loop indefinitely, never reaching `untilHeight`.

  Because `(*Syncer).collectHeaders` occurs before much of the validation, the attacker can construct the chain without having to include valid messages or proving storage. Also, because the chain is invalid, any compliant peer will respond with a failure status, so the peer list will be iterated through until reaching the malicious peer, and the malicious peer will be consistently queried for each chunk of blocks.
  Because there can only be `syncWorkerCount = 3`[6] syncing jobs performed concurrently, the malicious actor can repeat this 3 times (with different peers) to halt the node's syncing for an arbitrary duration.
- In `(*Syncer).syncFork` at `sync.go:1154`, a malicious height value can affect the loop's progress, but it will eventually exit.

Apart from the "endless sync" attack scenario, these issues do not otherwise appear abusable on their own. See LOT-07 for explanation of an attack that can benefit from the unchecked height values.

Similarly, there does not appear to be any validation that the height is positive. Because `abi.ChainEpoch` is defined as an `int64`, it is possible for the field to hold a negative value.

Recommendations

Unless there is benefit in allowing negative `abi.ChainEpoch` values (perhaps to distinguish between uninitialized values and the genesis epoch?), consider converting this to the more appropriately typed `uint64`.

Ascertain whether there is reason for the height to decrease by more than one along a chain (e.g. if it is possible for there to be an empty epoch during which no block was proposed). If this is possible, validate that each `TipSet` in a chain is strictly decreasing in height; otherwise ensure it strictly decrements.

[6] Defined at `lotus/chain/sync_manager.go:55`

Ensure this validation occurs early during processing of a `BlockSyncResponse`, before the height value is relied upon in `sync.go`.

Resolution

`(*BlockSync).processResponse()` validates the chain using `(*TipSet).IsChildOf()`. As well as checking that the `ts.Parents()` CIDs correspond to the next chain entry, it checks that the reported Height is strictly decreasing.

This issue was resolved as part of PR #2715.

LOT-11	Unlimited Active Peer Connections Allows DoS		
Asset	`lotus/chain/blocksync/blocksync_client.go` , `lotus/lib/peermgr/peermgr.go` & `go-fil-markets/storagemarket/impl/connmanager/connmanager.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

There are no limits to the number of peers allowed to connect to a Lotus node. Because maintaining active peer connections consumes resources, a malicious actor could generate lightweight peers to spam connections and starve the Lotus node of available network resources and memory.

If hardware limits are reached, the node would likely have difficulty making its own peer connections or participate in market deals.

Although the peer manager defines a `MaxFilPeers` value, no restrictions are currently in place; only a log is emitted when the peer limit is exceeded:

```
118     } else if pcount > pmgr.maxFilPeers {
          log.Debug("peer count about threshold: %d > %d", pcount, pmgr.maxFilPeers)
        }
```

peermgr.go

Similarly, there appears to be no restrictions on the number of Deal connections. If so, this can equivalently exceed available network resources.

Recommendations

Avoid allowing an unlimited number of peers to connect to a Lotus node.

Note that setting a naive hard limit to the number of active peers may introduce susceptibility to eclipse attacks.

One possibility would be to implement a soft limit and a buffer zone, where new peers are accepted while `pmgr.getPeerCount < MaxFilPeers + buffer`. However, after exceeding the soft limit, low-ranked peers are removed until peers are within an acceptable limit.

Because new peers are accepted after exceeding the limit, a malicious actor would be unable to prevent friendly peers from joining, thus breaking an eclipse attack. Also note that the priority given to new peers would need to be carefully considered, as spamming new peers could cause all friendly peers with a rank lower than the default priority to be evicted.

Also consider implementing IP-based blacklisting to increase the cost of peer spamming.

Resolution

Investigation revealed that while active lotus peers are not explicitly limited in `peermgr.go`, the number of overall libp2p network connections are limited by the `Libp2p.ConnMgrHigh` config value described in `node/config/def.go`.

This is passed to the `go-libp2p-connmgr` `BasicConnMgr`, which prunes connections once they exceed the configured limit. This is sufficient to resolve DoS due to a flood of new connections.

See Issue #3286 for context.

LOT-12	Withdraw Other Users Balance from the Market Actor		
Asset	`specs-actors/actors/builtin/market/market_actor.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `MarketActor` account facilitates deals between clients (storage users) and miners (storage providers). The `MarketActor` contains an escrow to store a quantity of tokens for each user which acts as collateral and payment for each deal.

In order to publish a deal both users must have the required funds deposited in the market.

Users are able to withdraw funds that are not locked by a deal at anytime by calling `WithdrawBalance()`, passing an `Address` as a parameter. `WithdrawBalance()` will transfer their funds from the `MarketActor` back to the given `Address`.

`WithdrawBalance()` calls `escrowAddress()` to validate the caller address. However, in the case where the nominal address (address where funds are withdrawn) is `CallerTypesSignable`, there is no validation that the caller address matches the nominal address.

This allows any user to withdraw funds on behalf of another user, so long as the nominal account and caller account are both of type `CallerTypesSignable`.

As the funds are transferred back into the nominal address, the funds are not stolen. This has the potential to be a DoS attack as any user may call `WithdrawBalance()` on behalf of another user which will prevent them from entering into deals due to insufficient funds to satisfy collateral requirements.

Note this attack will not work on `MinerAccountActors` as they require the calling address to be the owner or worker address of the miner.

Furthermore, a similar but less effective attack can be used on `AddBalance()`. In the case of `AddBalance()`, the funds deposited are attached to the `Value` of the message. As a result, the attacker would be paying to deposit funds on another user's behalf.

Recommendations

We recommend modifying the check in `escrowAddress()` to ensure that the caller is the nominal address for the cases where the nominal address is of type `CallerTypesSignable`.

Resolution

The issue has been resolved in the following PRs #718 and #1083.

LOT-13	Swapping a Signer in `MultiSigActor` Allows for a Signer to Approve a Transaction Twice		
Asset	`specs-actors/actors/builtin/multisig/multisig_actor.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A `MultiSigActor` allows the signers (if they agree) to swap a signer's address for another address.

The addresses are not modified for pending transactions. Hence, if there is an existing transaction which has the user's first address in the `Approved` array. They may call `Approve()` again after the address has been swapped which will add the new address to `Approved`.

A signer may repeat this process multiple times giving them the potential to be the only signer for a transaction if their `SwapSigner()` transactions are approved.

Recommendations

Consider iterating through existing `PendingTransactions` and swapping the addresses or otherwise prevent the double approving of a transaction.

Resolution

This issue has been resolved in PR specs-actors#1207, in which approvals for pending transactions are removed when their signer is removed (or swapped out).

LOT-14	`StorageMarket` `ConnManager` Unbound Number of Streams		
Asset	`go-fil-markets/storagemarket/impl/connmanager/connmanager.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The connection manager (`ConnManager`) adds streams on each Deal that is received and disconnects the stream if there was no valid response from the client. Due to the connections being kept in a map of `CID` , it may be possible to generate a number of valid `CID` s and use this to fill connections and use memory resources on the provider machine, causing a crash or stalling progress.

Recommendations

Similar to eviction of peers in the Lotus validation and syncing, the connection manager should provide eviction rules and limits to the number of active streams stored in the map.

Resolution

This issue has been sufficiently mitigated in PR go-fil-markets#297. While `go-fil-markets` does not explicitly enforce limits to the number of active connections (unlike the connection pruning `go-libp2p-connmgr` that prevents LOT-11), incoming deal connections are now short-lived in all cases such that a flood of deals would generally be processed more quickly than they can be transmitted. Prior to PR #297, connections for acceptable deals would remain open for the lifespan of the deal.

See Issue go-fil-markets#377 for context.

LOT-15	Deterministic Peer Ordering Can Be Abused		
Asset	`lotus/chain/blocksync/blocksync_client.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

While not a problem on its own, this can exacerbate other issues, like LOT-04, LOT-09, LOT-08, and LOT-07.

As part of a targeted attack, it is possible for a malicious actor to reliably control the first peers queried by the Syncer. This can be achieved by creating peers located geographically near the target, which will have a low latency and be preferred by `(*bsPeerTracker).prefSortedPeers()`.

In the case of LOT-07 and LOT-08, only by consistently being the first peer queried can a malicious actor effectively perform DoS and eclipse attacks.

Also, because peers are queried synchronously and new peers are initially ranked better than average due to a `newPeerMul < 1` (`newPeerMul = 0.9`), a malicious actor can impact syncing performance. By continuously spamming the node with new peers, the actor can provide multiple greatly delayed failure responses. These slow peers would be demoted, but only after the initial slow response has done its damage and fresh, malicious peers would take their place.

Recommendations

Exercise caution when providing higher-ranked peers unnecessary trust.

It may be worth considering ordering peers in wider “bands”, where members of the same band or score have the same priority, and are iterated through non-deterministically.

Ensure peers are demoted or evicted for malicious behavior (such as providing invalid or malformed responses), so highly ranked peers can only abuse their position once before being demoted. It is important that failure at more expensive levels of validation also results in peer demotion or eviction.

Resolution

This issue has been acknowledged by the development team but is part of a wider discussion of ranking peers and will be addressed when they implement the new peer ranking system.

LOT-16	`SettleDelay` is too Low in `PaychActor`		
Asset	`specs-actors/actors/builtin/multisig/multisig_actor.go`		
Status	**Resolved:** Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `SettleDelay` is used by `PaychActor` to give users enough time to submit their vouchers before `Collect()` can be called, which finalises and pays out the users.

The `SettleDelay` is currently set to one epoch. Users may therefore miss out on submitting their vouchers (assuming no other vouchers have a larger `MinSettleHeight`).

Recommendations

Consider increasing the `SettleDelay` to a value large enough that users may submit the vouchers comfortably after `Settle()` has been called.

Resolution

The settlement delay has been increased to 12 hours in commit 0d2ebd3.

LOT-17	Early Deal Termination from Sector Expiry		
Asset	`spec-actors`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

At the end of each epoch, the `CronActor` first calls `PowerActor.OnEpochTickEnd()`, which will eventually call `MinerActor.handleProvingPeriod()`. Any Sectors which have expired will terminate deals through the `MarketActor` by calling `requestTerminateDeals()`.

Terminated deals will burn the collateral of the miner.

If the `deal.EndEpoch = sectorExpiration` passes `MarketActor.validateDealCanActivate()`, then the `CronActor` will first call `PowerActor.OnEpochTickEnd()`, which will expire the sector and terminate deals before the `CronActor` calls `MarketActor.CronTick()` to successfully process the deal end. Hence, the miner's collateral will be burnt.

Furthermore, if `deal.EndEpoch < sector.Expiry` and all blocks between `deal.EndEpoch` and `sectorExpiry` are null blocks. Then `PowerActor.OnEpochTickEnd()` will be called before `MarketActor.CronTick()` causing the sectors to be terminated and the miner collateral burnt.

Recommendations

We recommend enforcing `deal.EndEpoch < sectorExpiry` and consider adding a small buffer to compensate for potential null blocks.

Resolution

The codebase has been updated such that this issue is no longer relevant as the miner cron no longer expires sectors and on-time expiration does not terminate deals.

LOT-18	Sector Expiration not Validated in `ExtendSectorExpiration()`		
Asset	`specs-actors/actors/builtin/miner/miner_actor.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `ExtendSectorExpiration()` does not check `params.NewExpiration` when the sector should expire which should be "exactly *the epoch before* the start of a proving period".

Furthermore, there is the potential for overflow on line [578] in `extensionLength := params.NewExpiration - oldExpiration` if `parmas.NewExpiration` is set to a large negative number.

Recommendations

Consider implementing sufficient validation on the `NewExpiration` to prevent overflows and to prevent the ending not aligning with a proving period.

Resolution

The codebase has been updated such that this issue is no longer relevant.

LOT-19	Generated `UnmarshalCBOR` Methods Allow Omission of Struct Fields		
Asset	`whyrusleeping/cbor-gen/gen.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Generated `UnmarshalCBOR()` methods, which involve unmarshalling CBOR Map types to structs, will return without error when passed CBOR bytes that do not contain relevant struct fields i.e. the method does not require that all struct fields be present in the input.

The methods, generated by `emitCborUnmarshalStructMap()`, simply iterate through the Map items and assign them to their corresponding field. If a struct field is missing, no error is emitted and the resulting struct has that field left as the type's zero-value.

While this behavior may initially appear harmless, it can cause some complications, particularly in combination with other issues:

- When the omitted field is an `Address` (defined in `filecoin-project/go-address/address`), the resulting struct can contain an `address.Undef` even though the implementation of `(*Address).UnmarshalCBOR` illustrates an intention to prevent the unmarshalling of an undefined address. The Map implementation effectively bypasses the intended validation, and can break assumptions made elsewhere.[7]

  Because the marshalling of an `address.Undef` is forbidden, this enables some CBOR-encoded bytes to unmarshal without error, but the result will fail to marshal.

  - This currently affects the `ChannelInfo` and `PaymentInfo` types, defined in `lotus/paychmgr` and `lotus/api` respectively.[8]
  - If the `Message` type[9] used `MajMap` as part of its CBOR encoding, this would break the assumption made in `(*Message).CID()` (`message.go:107`) that *"we must be able to marshal something that was successfully unmarshalled"*. Untrusted network input could then trigger a panic.
  - Similarly, this could cause problems where errors are returned when trying to persist CBOR-encoded VM state to disk.

- This means different CBOR bytes can unmarshal to the same struct.

  If a `MajMap` were used as part of encoding a `BlockHeader` and a miner proposed a block containing a zero-value, a malicious actor could submit the alternative encoding (with the zero-value omitted) to `VerifyConsensusFault()`[10], causing the miner to be slashed for "double signing".

  NOTE: This is possible due to a weakness in the current `VerifyConsensusFault` implementation, see LOT-01

[7] This would similarly affect any other type where the implementation forbids the marshal/unmarshal of a zero-value.

[8] Fuzzing results, located at `fuzz_build/paymentinforaw/crashers/crash-999e152ff15c2ebe71f7b3207f77bdc0980faab4` and `fuzz_build/channelinforaw/crashers/crash-5bb788680de54d757405c2ac2c91a5a708aa88d4` are representative examples.

[9] Defined in `lotus/chain/types/message.go`

[10] Defined in `lotus/chain/vm/syscalls.go`

While the testing team has not identified any currently exploitable issues during the time allocated to this assessment, such behavior is likely to cause unintended behaviour.

Recommendations

Consider adjusting `emitCborUnmarshalStructMap()` to check that all of the struct's fields are present in any CBOR input. One implementation could achieve this by verifying that the number of CBOR Map items are equivalent to the number of struct fields, and that no duplicate Map keys are present.

Ensure all resulting code is updated to reflect any changes made to the code generation.

Resolution

Omission of CBOR fields is intentionally allowed in order to make migration and updates easier (allowing backwards-compatible changes to the messages).

It may be possible to protect against the current behaviour involving `Address.Undef` by creating an appropriate zero-value CBOR and passing it to the field's `UnmarshalCBOR()` method whenever the field is omitted in the raw CBOR. However, the implementation complexity has been deemed to outweigh the benefit.

This issue has been acknowledged by the development team and marked as *wontfix*.

LOT-20	No Bounds Checks for `(*Int).MarshalCBOR()`		
Asset	`specs-actors/actors/abi/big/int.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Unlike with `(*Int).UnmarshalCBOR()`, `(*Int).MarshalCBOR()` does not validate that the result is within `BigIntMaxSerializedLen` (128 bytes).

Because of this, it is theoretically possible for a Lotus node to marshal a value that it cannot unmarshal.

The testing team have not identified any directly exploitable attack vectors due to this issue, but such a situation could be troublesome when saving VM state to the datastore (subsequent attempts to retrieve that datastore item would fail).

Recommendations

Consider returning an error if the output of `(*Int).MarshalCBOR` exceeds `BigIntMaxSerializedLen` in size.

Resolution

A bounds check has been added when marshalling `Int` to ensure that it does not exceed the maximum CBOR length, in PR #859.

LOT-21	Validation of Block Message Roots Persists Data to Storage		
Asset	`lotus/chain/sync.go`		
Status	**Closed:** See Issue lotus#3320		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

As part of validating a new block, the Lotus node computes Trie roots of the block's messages and verifies that those roots match the block header's `Messages` field. This validation occurs in `zipTipSetAndMessages()` and `(*Syncer).ValidateMsgMeta()`, which use `computeMsgMeta()` to calculate the root.

However, this Trie structure is constructed inside an IPFS key-value blockstore which, for `ValidateMsgMeta()`, is the `Syncer`'s main `ChainStore` and is persisted to disk.

`iterFullTipsets` passes `zipTipSetAndMessages` a temporary, in-memory `IpldStore` so, on error, no data is persisted. However, after processing is successful, the AMT structure created by `computeMsgMeta` appears to be unnecessarily persisted with the rest of the blocks and messages.

Because there does not not appear to be any garbage collection present to remove unneeded items from a `BlockStore`, these wasted resources are effectively *lost*.

The impact is likely small when compared to overall resource utilisation, but will increase running costs and may affect storage miners' margin estimates.

A malicious actor may broadcast invalid blocks which, via `ValidateMsgMeta()`, will persist data even though the block is invalid and otherwise thrown away. By slowly increasing competing miners' overheads, the actor could increase their own profitability at the expense of the network.

Recommendations

If an `(ipfs/go-ipld-cbor).IpldStore` is needed to verify the `Messages` root, avoid persisting the Trie structure to disk, particularly for blocks that are found to be invalid.

In-memory caching may be useful to save compute resources when blocks share a subset of messages, but long-lived persistence in memory should also be avoided.

LOT-22	`ReadByteArray()` Not Using `maxlen` Argument to Reject Oversized Buffers		
Asset	`whyrusleeping/cbor-gen/utils.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `ReadByteArray` function is passed a `maxlen` argument, however, is using a hard-coded value of `256*1024` to reject input that is too long.

Line [469] consists of a condition to return an error for strings greater than the maximum length, but is not referencing the provided maximum length.

```
func ReadByteArray(br io.Reader, maxlen uint64) ([]byte, error) {
460   maj, extra, err := CborReadHeader(br)
  if err != nil {
462     return nil, err
  }
464
  if maj != MajByteString {
466     return nil, fmt.Errorf("expected cbor type 'byte string' in input")
  }
468
  if extra > 256*1024 {
470     return nil, fmt.Errorf("string in cbor input too long")
  }
472
  buf := make([]byte, extra)
474   if _, err := io.ReadAtLeast(br, buf, int(extra)); err != nil {
    return nil, err
476   }

478   return buf, nil
}
```

whyrusleeping/cbor-gen/utils.go

Recommendation

Update line [469] to use `if extra > maxlen` rather than the hard-coded value.

This recommendation is to be applied in the upstream library of `whyrusleeping/cbor-gen/utils.go` to fix all dependent downstream packages.

Resolution

The hard coded value of `256*1024` is replaced with the variable `maxLen` in PR #43.

LOT-23	Potential to Reduce `ToSend` in `PaychActor`		
Asset	`specs-actors/actors/builtin/paych/paych_actor.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `PaychActor` is in charge of managing the transfer of funds between two parties using vouchers. The payee's (receiver of funds) balance is stored in the state variable `ToSend`.

The `ToSend` amount is the total received from all lanes.

During a lane merge, the amount redeemed from other lanes is summed into `redeemedFromOthers` and the amount to be redeemed from the current lane is `ls.Redeemed`.

The line below takes the current amount to be redeemed, less the amounts already redeemed across all merged lanes. There is no guarantee that `balanceDelta` is greater than zero.

```
balanceDelta := big.Sub(sv.Amount, big.Add(redeemedFromOthers, ls.Redeemed))
```

If `newSendBalance := big.Add(st.ToSend, balanceDelta)` is greater than or equal to zero, the function will not revert.

That is if `ToSend >= balanceDelta`, it will set `st.ToSend = newSendBalance` which would reduce the `ToSend` value.

Recommendations

We recommend enforcing that `balanceDelta >= 0` to prevent the decrement of the `ToSend` balance.

Resolution

The issue is a deliberate design feature of the payment channels. It allows users to split gas costs by merging channels (which uses less gas than redeeming vouchers individually) at an amount less than the total redeemed.

See the issue #1046 for the full discussion.

LOT-24	API Call to `ChainStatObj` Crashes Daemon		
Asset	lotus/node/impl/full/chain.go		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `ChainStatObj` API function returns information relevant to the specified chain object. If this API function is called using the below `curl` command, where the arguments are of type `null`, the caller is able to cause a panic causing the Lotus daemon to crash. Additionally, the user only needs `read` permissions on their JWT token to cause this crash.

```
curl -X POST \
-H "Content-Type: application/json"  \
-H "Authorization: Bearer $(cat ~/.lotus/token)" \
--data '{ "jsonrpc": "2.0", "method":"Filecoin.ChainStatObj", "params": [null, null], "id
": 0 }' \
'http://127.0.0.1:1234/rpc/v0'
```

The `panic` can be traced to `cid.go` in the `go-cid` dependency. The issue occurs when an invalid `c Cid` is parsed to the `Prefix()` function in `chain.go` line [206] when there is an assumption that `c.Hash()` will successfully return a valid multiaddr to the `Decode()` function in `cid.go` line [517].

Due to the fact that the API is only exposed to the *localhost* interface, this issue is raised with a *low* severity.

Recommendation

There are two ways this issue can be solved:

- Allow `Prefix()` to return an error if the `Decode()` function receives an invalid multiaddr;
- Implement prior validation for the `ChainStatObj()` API function to ensure that `c Cid` is successfully decoded.

The testing team recommends implementing the second option.

Resolution

This issue was resolved in `go-cid` commit f7cb4c91e.

Calling `Prefix()` with an invalid CID may return garbage, but it will no longer cause a nil pointer dereference.

LOT-25	Transfers to Certain Actors Results in Lost Funds
Asset	`lotus/chain/vm/vm.go`
Status	Open
Rating	Informational

Description

Messages which contain a positive `Value` automatically invoke the `transfer()` function in the virtual machine which results in funds being moved between accounts.

There are no restrictions on the `To` parameter in a message. Therefore, it is possible to transfer funds to any actor. Funds can be therefore be transferred to all builtin actors which if the `Method` is set to 0 will result in a successful execution.

Some builtin actors such as `MultiSigActor` and `PaychActor` are designed to have funds directly transferred.

However, funds may be lost or undefined results may occur when transferring funds to the `SystemActor`, `RewardActor`, `gasHolderAccount`, `CronActor` or others.

Recommendations

Consider restricting the `To` address to certain actor account IDs for messages with `Method = 0`.

Resolution

This issue has been acknowledged and raised for discussion as github issue #1069.

LOT-26	Validation of Piece Sizes
Asset	`lotus/chain/vm/syscalls.go`
Status	Open
Rating	Informational

Description

The function `ComputeUnsealedSectorCID()` sums the size of individual pieces and then discards the sum without using it again.

There is the potential for this sum to overflow. Additionally, each piece should be specifically checked to be less than the sector size.

Recommendations

We recommend adding checks that each piece size is less than or equal to `RegisteredProof.SectorSize()` and that the total sum of piece sizes is less than `RegisteredProof.SectorSize()`.

Resolution

This issue has been raised for discussion as GitHub issue #3222.

LOT-27	`VerifiedRegistryRoot` Address is Constant
Asset	`lotus/chain/gen/genesis/genesis.go`
Status	Open
Rating	Informational

Description

The `VerifiedRegistryRoot` address is set to have a constant ID `80` in `MakeInitialStateTree()` line [221]. For consistency and to prevent accidental overwriting, this address should be registered in `specs-actors/actors/ builtin/singletons.go`.
Additionally it should be given a code in a similar style to other builtin actors (as seen in `specs-actors/actors/builtin/codes.go`).

Recommendations

Consider adding the `VerifiedRegistryRoot` account as a builtin account in `spec-actors`.

Resolution

This issue has been acknowledged and raised for discussion as github issue #1031.

LOT-28	`MultiSigActor.Constructor()` Parameter Verification
Asset	`specs-actors/actors/builtin/multisig/multisig_actor.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `MultiSigActor` constructor takes an array of `Signers` as input. There are no checks to ensure that duplicates are not added to this list.

The testing team did not identify during the allocated time any direct exploitation scenario from having a duplicate signer.

The input to `Constructor()` also includes `NumApprovalsThreshold`. However, there is no bounds check to ensure this is an obtainable number. For example it may be set to a negative number or set higher than the number of signers. This could make the account unusable or callable by anyone.

Additionally, there are no checks on `UnlockDuration` such that it may be set to a negative number.

Recommendations

Consider adding the following checks for each of the parameters in the `Constructor()` to ensure a valid initial state for the account:

- No duplicates in `Signers`;
- `0 < NumApprovalsThreshold <= len(Signers)`;
- `UnlockDuration >= 0`.

Resolution

The parameter validation checks described above have been added in PRs #443 and #912.

LOT-29	`MultiSigActor` Can Manipulate `NumApprovalsThreshold` to an Invalid State
Asset	`specs-actors/actors/builtin/multisig/multisig_actor.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `RemoveSigner()`, if passed `Decrease = true`, will always decrement the `NumApprovalsThreshold` by one, irrelevant of its current value. Thus, if the current value is one it will be decremented to zero and from zero it will be decremented to a negative one.

Setting the value to less than one is an invalid state and should be avoided.

Recommendations

We recommend adding a check in `RemoveSigner()` to ensure the value does not drop below one.

Resolution

A check has been added to prevent the threshold from falling below one in PR #1156.

LOT-30	Potential `nil` Pointer Access in `ReportConsensusFault()`
Asset	`specs-actors/actors/builtin/miner/miner_actor.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The function `ReportConsensusFault()` calls `rt.Syscalls().ReportConsensusFault()` which may return `(nil, nil)` if there are no errors and no consensus faults.

As a result the following call, `fault.Epoch`, will raise a nil pointer exception.

Recommendations

Consider handling the case where the return value of `Syscalls().ReportConsensusFault()` is `(nil, nil)`.

Resolution

The codebase has been modified such that this issue is no longer relevant.

LOT-31	`BlockSync.processBlocksResponse` Does Not Verify Length
Asset	`lotus/chain/blocksync/blocksync_client.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Upon receiving a set of blocks, the call to `processBlocksResponse` does not verify that `len(response.Chain) <= req.RequestLength` (for any `BlockSyncResponse response` and its associated `BlockSyncRequest req`).

Although no immediate security implication, there is potential wasted computation and time converting and validating unnecessary `BSTipSet` entries to `TipSet` for the length of the chain received.

Recommendations

The length of `response.Chain` should match the `req.RequestLength` when `response.Status == StatusOK` and `len(response.Chain) < req.RequestLength` when `response.Status == StatusPartial`.

The testing team are not aware of any situation where a compliant node would respond with a chain greater than the requested length, so it appears to be safe to treat such responses as malformed.

Resolution

This was resolved as part of PR #2715.

`(*BlockSync).processResponse()` validates the length of successful and partial responses.

LOT-32	Miscellaneous Observations in the Lotus & Specs-Actors Codebases
Asset	`lotus` & `specs-actors`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team on the Lotus codebase that do not have direct security implications.

- In `lotus/chain/sub/incoming.go` there are a two unhandled errors:
  - line [115]: `lru.New2Q` unhandled error.
  - line [336]: `lru.New2Q` unhandled error.
- Package import shadowed by variable assignment could cause confusion:
  - `lotus/chain/sub/incoming.go:297` : **state** overshadows package state.
  - `lotus/chain/stmgr/stmgr.go` :
    - line [342]: **state** overshadows package import state.
    - line [352]: **state** overshadows package import state.
    - line [453]: **state** overshadows package import state.
- In `lotus/chain/types/actor.go:13` the comments reference `chain/actors/actors.go` which does not exist.
- The type `Message` has `GasLimit` as `int64` which is always greater than or equal to zero and regularly converted to `uint64` .
- In `lotus/chain/vm/mkactor.go:52` the `PricelistByEpoch()` function is called. The results are already available in `rt.pricelist` .
- In `lotus/chain/vm/mkactor.go` new BLS addresses are not guaranteed to be valid points.
- In `lotus/chain/vm/invoker.go` the error descriptions are not accurate:
  - line [119]: `return nil, newErr("second argument should be Runtime")` should be `"second argument should be a pointer"` .
  - line [123]: `return nil, newErr("wrong number of outputs should be:  " + "cbg.CBORMarshaler")` should delete `+ "cbg.CBORMarshaler"` as that is part of the next check.
- In `lotus/chain/vm/runtime.go` the functions `Create()` and `Transaction()` will fail silently if `stateCommit()` fails.
- In `specs-actors/actors/builtin/market/market_actor.go:78` the variable `amountSlashedTotal` is always set to zero. The check on line [106] is therefore unnecessary.

- In `specs-actors/actors/builtin/market/market_state.go:278` there is a call `st.GetEscrowBalance(rt, addr)`. This call is unnecessary as the value is already available as `escrowBalance` on the previous line.
- In `specs-actors/actors/builtin/verifreg/verified_registry_actor.go` the function `AddVerifier()` does not check `params.Allowance >= 0`.
- In `specs-actors/actors/builtin/miner/miner_actor.go:L108` the check `Assert(periodStart > currEpoch)` is already performed in `nextProvingPeriodStart()`.
- There are a range of constants that are labelled `PARAM_FINISH` which need to be carefully selected.
- In `specs-actors/actors/builtin/miner/miner_actor.go` the function `verifyPledgeMeetsInitialRequirements()` is not implemented.

Resolution

The development team has acknowledged and understood the miscellaneous issues described above.

LOT-33	Miscellaneous Observations in Dependencies
Asset	`filecoin-project/`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details any miscellaneous findings discovered by the testing team in the dependencies listed in the **Introduction**. These findings do not have any direct security implications.

`filecoin-project/go-cbor-util`

- **README.md:42** Incorrect example of `CBORUnmarshaler` in `README` *Usage*.
  - `var out cbg.CBORUnmarshaler` should be `var obj cbg.CBORUnmarshaler`.

`filecoin/project/go-fil-commcid`

- **README.md** has function names misspelt.
  - line [48]: `PieceCommmitmentV1ToCID(commP)` change to `PieceCommitmentV1ToCID(commP)` (remove third "m").
  - line [49]: `DataCommmitmentV1ToCID(commD)` change to `DataCommitmentV1ToCID(commD)` (remove third "m").
  - line [50]: `ReplicaCommmitmentV1ToCID(commR)` change to `ReplicaCommitmentV1ToCID(commR)` (remove third "m").
  - line [95]: `CommmitmentToCID(commIn, hashingAlgorithm)` change to `CommitmentToCID(commIn, hashingAlgorithm)` (remove third "m").

Resolution

The development team has acknowledged and understood the miscellaneous issues described above.

LOT-34	Unmarshalling Signatures may Panic		
Asset	`specs-actors/actors/crypto/signature.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Signatures are converted from bytes to a go object in the function `signature.UnmarshalCBOR()`. The first byte of the input is a header byte which contains the length of the remaining bytes. If the length, `l`, is set to zero the function will create a buffer of length zero. The buffer will then be indexed at `0`, causing an index out of bounds panic when the length is zero.

```
buf := make([]byte, l)
if _, err = io.ReadFull(br, buf); err != nil {
  return err
}
switch SigType(buf[0]) {
```

signature.go

Signatures are an essential tool in Lotus and are widely used, not just in the core VM logic but also in other modules, such as `go-data-transfer` and `go-fil-markets`. As a result this panic is accessible from a range of locations in the codebase.

Recommendations

Consider returning an error in the case where the length is zero.

Resolution

An additional check to ensure that the length is greater than zero has been added in PR #1103.

LOT-35	Storage Miner Custom Deal Filter Logic		
Asset	`lotus/node/modules/storageminer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Miners or storage providers set custom deal filtering logic to prevent deals which do not fit the current requirements of the storage miner.

An example of deal filtering logic is that the deal `StartEpoch` must be sufficiently far in the future to allow the miner time to seal the sector.

The implementation of the custom deal logic can be seen in the function `BasicDealFilter()`. The function `BasicDealFilter()` does not restrict the maximum value of `StartEpoch`. As a result `StartEpoch` can be set excessively far into the future.

Fees are paid to the miner by multiplying the price by the duration which is `EndEpoch - StartEpoch`. The miner will store the piece from when the deal is accepted in the present until `EndEpoch`.

Therefore the miner will not receive payment for storing the piece between the present and `StartEpoch`. Thus, clients can reduce the cost of storing an item by increasing the `StartEpoch`.

Note deals have a minimum duration based off their sector size so the client will have to pay at least `minDuration * price`.

Recommendations

We recommend restricting the `StartEpoch` to within a reasonable bound which allows the miners sufficient time to seal the sector and add new deals but limit the amount of time they are storing data without payment.

Resolution

This was resolved in PRs #4173 and #4337.

The default deal filter `BasicDealFilter()` rejects deal proposals that have a `StartEpoch` greater than 7 days from $now + sealEpochs$. Storage miners can still implement custom deal filters allowing more finer-grained control.

LOT-36	Storage Market `DealProposals` Validation
Asset	`go-fil-markets/storagemarket/impl/providerstates/provider_states.go`
Status	Open
Rating	Informational

Description

Storage market `DealProposals` are a contract between a client, wishing to store data, and a provider, who is able to store the data. Within these contracts is all information related to the data to be stored, the amount paid for storage and how long it should be stored for. If both users are happy with the deals they will be published on-chain to the `StorageMarketActor`.

The client sends a deal proposal to a provider who validates the deal before accepting it. It is validated in `ValidateDealProposal()`. However, `ValidateDealProposal()` does not provide sufficient checks to ensure that the deal can be successfully published on-chain.

Deal proposals are given a start and end date (`StartEpoch` and `EndEpoch` respectively), but the duration `EndEpoch - StartEpoch` is not verified to be within the `StorageMarketActor` duration requirements.

Similarly, the available balance of the client in the `StorageMarketActor` is checked to ensure they have sufficient funds for the proposal. The check to ensure the client has sufficient balance is to ensure the client available balance is greater than `proposal.TotalStorageFee()`, which does not account for the client collateral requirements. Note that the default client collateral requirements is zero. The function `proposal.ClientBalanceRequirement()` should be used as it encompasses the client collateral requirements too.

This issue is raised as informational as any invalid deals will be caught before publishing on-chain hence will not cost the miner transaction fees for publishing the deals.

Recommendations

We recommend implementing as much of the on-chain deal validation logic in `ValidateDealProposal()` such that deals are rejected as early as possible.

LOT-37	Wallet Keystore Secret Information is Stored Unencrypted
Asset	`lotus/chain/wallet/wallet.go`
Status	Open
Rating	Informational

Description

Accounts on Lotus are controlled public keys from either SECP256k1 or BLS12-381 curves. The private keys are used to sign transactions and send FIL. A compromise of the private keys results in a full compromise of the account.

The private keys of the accounts are stored unencrypted in the filesystem.

Any attacker with permission to read as the user will now have full access to the private key information.

Note this falls within the threat model of Lotus which states that if an attacker gains access to the machine that is considered a full compromise, thus is raised as informational.

Recommendations

We recommend encrypting the private key field in the keystore to improve the resistance to attack. This can open up possibilities to allow the Lotus node operator to provide the decryption key in a more resilient fashion (e.g. interactively, or via a secrets manager or HSM).

LOT-38	JSON Web Token is Stored Unencrypted
Asset	`lotus/api`
Status	Open
Rating	Informational

Description

The Lotus API is permissioned via JWTs. Each JWT is given a permission level of read, write or admin, which allows certain API functions to be called.

A default JWT token is created with admin permission. This JWT is used by the command line functions to interact with the Lotus daemon.

This JWT is stored unencrypted in the filesystem. Thus, any attacker with the users read permission may copy this JWT and make admin-permissioned API calls such as `WalletExport`, extracting the private key of the wallet.

Note this falls within the threat model of Lotus, which states that if an attacker gains access to the machine that is considered a full compromise, thus is raised as informational.

Recommendations

We recommend giving users the option to password encrypt the default JWT.

LOT-39	Retrieval Path is Local to Daemon not Caller
Asset	`lotus/cli`
Status	Open
Rating	Informational

Description

Storage clients may retrieve a file that has been stored by a miner by using the CLI command `lotus client retrieve <CID> <output_path>`. The CLI command connects to the daemon via an API which then retrieves the file.

If the `output_path` is a local path rather than full file path, the output will be saved to the local path based on the location of the daemon process rather than the location of the caller process.

Recommendations

We recommend modifying the CLI command such that it converts a local path to a full path before calling daemon's API.

LOT-40	Client Imports Small Piece Sizes
Asset	`lotus/node/impl/client/client.go`
Status	Open
Rating	Informational

Description

A client will import data from a file to the Lotus daemon, which encodes the data into the piece format such that it may be sent to a provider for storage.

The piece size is rounded up to the nearest power of two as required by the storage formatting requirements.

There is a minimum size requirement for pieces that are stored on-chain. However, the client import does not enforce that pieces meet this requirement.

A piece will be successfully imported even if it does not meet the size requirements. If a client attempts to store this piece with a miner by making a deal proposal it will be rejected due to the piece being too small.

Recommendations

We recommend padding a piece when it is being imported such that the minimum size requirement is met.

Appendix A Fuzzing Harness

This section contains the information about the Fuzzing harness developed to support the identification of any issues relating to handling input from various sources. The fuzzer aims to identify any potential risks to public-facing elements that receive information over the network and are deserialized and processed.

The fuzzing harness structure consists of:

fuzz/*.go The `fuzz.go` file consists of both structured and unstructured fuzzing targets, using the go-fuzz fuzzing engine. The fuzzing performed aims at manipulating input to provide maximal code coverage.

fuzz/libfuzzer/* These utilize the `libFuzzer` fuzzing engine, which relies on experimental, builtin coverage instrumentation, introduced on Go1.14. This has the advantage of being able to successfully build packages relying on CGO and incorporating FFI (Foreign Function Interfaces).

The testing team primarily utilized this fuzzer to target the "CBOR" marshal and unmarshal that occurs throughout the codebase. Any `panic` observed on public-facing structs can be potential targets for malicious actors to send messages on the network and crash nodes.

Output

The fuzzing harness produces a number of output files, both `go-fuzz` and `libFuzzer` differing in format, but can be interpreted similarly. The output of the fuzzing harness can be viewed in the **fuzz_build** directory.

Corpus The corpus is the list of input that has been used throughout the fuzzing execution. It indicates the generated input and is used throughout execution to further manipulate fuzzing input in the hope of gaining more coverage and identifying issues.

Crashers The crashers indicate the input and output that has produced an error caught by the fuzzer. The `.output` files indicate the output that was produced, containing the message of the crash.

Suppressions The suppressions is the folder that contains output that has already been matched by the fuzzing harness. This reduces the number of duplicates that are reported.

This is not present for libfuzzer-based targets.

Coverage

The fuzzing harnesses currently target the following structures:

fuzz.go

- **BlockHeader**: `lotus/chain/types/blockheader.go`

- **BlockMsg**: `lotus/chain/types/blockmsg.go`
- **BlockSyncRequest**: `lotus/chain/blocksync/blocksync.go`
- **BlockSyncResponse**: `lotus/chain/blocksync/blocksync.go`

libFuzzer/

- **HelloMessage**: `lotus/node/hello/hello.go`
- **LatencyMessage**: `lotus/node/hello/hello.go`
- **VoucherInfo**: `lotus/paychmgr/store.go`
- **ChannelInfo**: `lotus/paychmgr/store.go`
- **PaymentInfo**: `go-fil-markets/retrievalmarket/types.go`
- **SealedRef**: `lotus/api/api_storage.go`
- **SealedRefs**: `lotus/api/api_storage.go`
- **SealTicket**: `lotus/api/api_storage.go`
- **SealSeed**: `lotus/api/api_storage.go`
- **Actor**: `lotus/chain/types/actor.go`
- **TipSet**: `lotus/chain/types/tipset.go`
- **SignedMessage**: `lotus/chain/types/signedmessage.go`
- **MsgMeta**: `lotus/chain/types/blockheader.go`
- **MessageReceipt**: `lotus/chain/types/message_receipt.go`
- **DealProposal**: `go-fil-markets/retrievalmarket/types.go`
- **SendParams**: `specs-actors/actors/puppet/puppet.go`
- **PublishStorageDealsParams**: `specs-actors/actors/builtin/market/market_actor.go`
- **VerifyDealsOnSectorProveCommitParams**: `specs-actors/actors/builtin/market/market_actor.go`
- **ComputeDataCommitmentParams**: `specs-actors/actors/builtin/market/market_actor.go`
- **OnMinerSectorsTerminateParams**: `specs-actors/actors/builtin/market/market_actor.go`
- **CreateMinerParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **DeleteMinerParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **EnrollCronEventParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **OnSectorTerminateParams**: `specs-actors/actors/builtin/power/power_actor.go`

- **OnSectorModifyWeightDescParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **OnSectorProveCommitParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **OnFaultBeginParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **OnFaultEndParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **MinerConstructorParams**: `specs-actors/actors/builtin/power/power_actor.go`
- **SubmitWindowedPoStParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **TerminateSectorsParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **ChangePeerIDParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **ProveCommitSectorParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **ChangeWorkerAddressParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **ExtendSectorExpirationParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **DeclareFaultsParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **DeclareFaultsRecoveredParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **ReportConsensusFaultParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **CheckSectorProvenParams**: `specs-actors/actors/builtin/miner/miner_actor.go`
- **ExecParams**: `specs-actors/actors/builtin/init/init_actor.go`
- **AddVerifierParams**: `specs-actors/actors/builtin/verifreg/verified_registry_actor.go`
- **AddVerifiedClientParams**: `specs-actors/actors/builtin/verifreg/verified_registry_actor.go`
- **UseBytesParams**: `specs-actors/actors/builtin/verifreg/verified_registry_actor.go`
- **RestoreBytesParams**: `specs-actors/actors/builtin/verifreg/verified_registry_actor.go`
- **ProposeParams**: `specs-actors/actors/builtin/multisig/multisig_actor.go`
- **AddSignerParams**: `specs-actors/actors/builtin/multisig/multisig_actor.go`
- **RemoveSignerParams**: `specs-actors/actors/builtin/multisig/multisig_actor.go`
- **TxnIDParams**: `specs-actors/actors/builtin/multisig/multisig_actor.go`
- **ChangeNumApprovalsThresholdParams**: `specs-actors/actors/builtin/multisig/multisig_actor.go`
- **SwapSignerParams**: `specs-actors/actors/builtin/multisig/multisig_actor.go`
- **UpdateChannelStateParams**: `specs-actors/actors/builtin/paych/paych_actor.go`
- **ModVerifyParams**: `specs-actors/actors/builtin/paych/paych_actor.go`
- **PaymentVerifyParams**: `specs-actors/actors/builtin/paych/paych_actor.go`
- **AwardBlockRewardParams**: `specs-actors/actors/builtin/reward/reward_actor.go`

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.

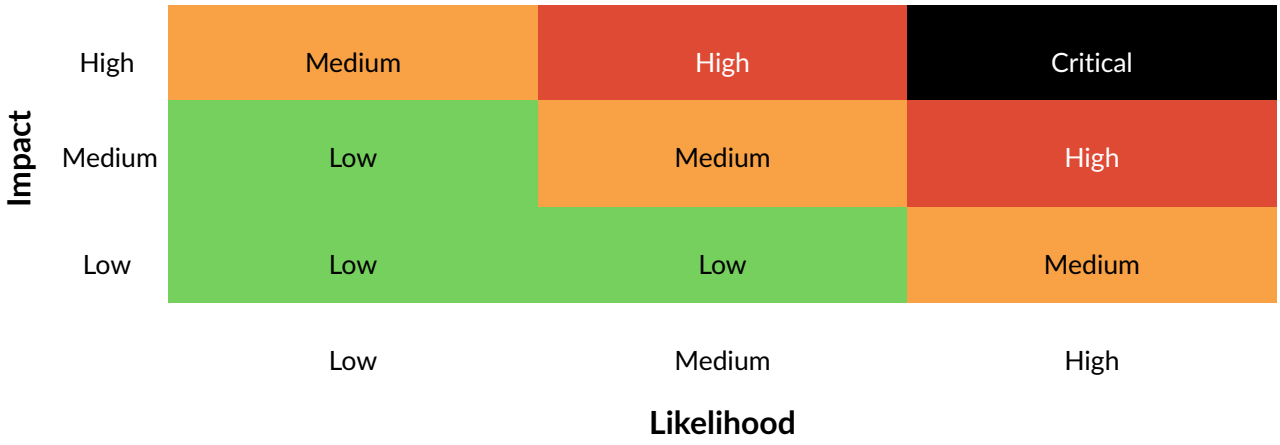

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References