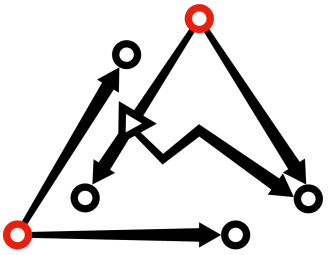

Least Authority

PRIVACY MATTERS

Lotus Implementation + Subcomponents

Security Audit Report

Protocol Labs

Final Report Version: 19 December 2020

Table of Contents

Overview

Background

Protocol Labs has requested that Least Authority perform a security audit of Lotus, an implementation of the Filecoin Distributed Storage Network, and its subcomponents, in preparation for the Filecoin mainnet launch.

Filecoin is a decentralized storage network that transforms unused cloud storage into an algorithmic market in which miners and clients are incentivized to participate. It leverages a token, Filecoin, to facilitate the negotiation of data storage and retrieval services. Miners earn filecoin, a native protocol token, by providing data storage and/or retrieval while clients pay miners for data storage or distribution and retrieval.

Project Dates

- **August 31 - September 30:** Initial Review *(Completed)*
- **October 7:** Initial Audit Report delivered *(Completed)*
- **November 23:** Updated Initial Audit Report delivered *(Completed)*
- **December 16-18:** Verification Review *(Completed)*
- **December 19:** Final Audit Report delivered *(Completed)*

Review Team

- Dylan Lott, Security Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Bryan White, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Lotus Implementation + Subcomponents followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Lotus Core: https://github.com/filecoin-project/lotus
- Markets
    - https://github.com/filecoin-project/go-fil-markets
    - https://github.com/ipfs/go-graphsync
- Storage Miner: https://github.com/filecoin-project/lotus/tree/master/miner
- Dependencies
    - github.com/filecoin-project/go-address
    - github.com/filecoin-project/go-amt-ipld
    - github.com/filecoin-project/go-bitfield
    - github.com/filecoin-project/go-cbor-util
    - github.com/filecoin-project/go-crypto
    - github.com/filecoin-project/go-data-transfer
    - github.com/filecoin-project/go-fil-commcid
    - github.com/filecoin-project/go-padreader
    - github.com/filecoin-project/go-sectorbuilder

- github.com/filecoin-project/go-statemachine
- github.com/filecoin-project/go-statestore
- github.com/ipfs/go-hamt-ipld
- github.com/ipfs/go-ipld-cbor
- github.com/whyrusleeping/cbor-gen
- github.com/ipld/go-ipld-prime

Third party code and the following components are considered out of scope:

- Anything that relates to the Filecoin protocol construction (known as Filecoin Theory). Examples of this are: Expected Consensus (EC), Network CryptoEconomics, and Proofs of Storage (PoST and PoR).
- Lotus does not implement its own cryptographic primitives as they are all imported from other libraries. Unless they are explicitly listed in the dependencies to review, they have been excluded from the scope of the audit.
- Proofs implementation, developed in Rust, were audited separately. The FFI is the boundary between the two and it is excluded from the audit. In particular, the https://github.com/filecoin-project/filecoin-ffi repository is considered the start of the exclusion zone.
- Dependencies such as IPFS, libp2p, and Drand are out of scope as they have been independently audited by other teams.
- Actors: https://github.com/filecoin-project/specs-actors

Specifically, we examined the following Git revisions for our initial review:

lotus@b8bbbf3ea3b186e658be9a8011fd6827b13aa3e5

go-fil-markets@80b1788108acd0664a7a1b89f9569ad6a59f821d

go-ipld-prime@350032422383277e6545b9b1a49112123b5c43fb

go-fil-commcid@8f644712406f0835267113151cf1aa7c18cc128b

go-padreader@548257017ca630a752df0776553ea459f8417293

go-graphsync@9529ffb39e7f5ec01ad973f4aec9e53152c96650

go-address@4490824631d6bdf7faf2ca857c67a07f3f90b814

go-amt-ipld@e559a05791617ca6f4c8429979a33c679690ec91

go-bitfield@a2cc0c7daec7b08fd9d7cb3152bd6caaa228cbd5

go-cbor-util@08c40a1e63a282cbe9ace616489357ff2f941b13

go-crypto@effae4ea9f030bfb05c3caaa42eb25bba317d5b7

go-data-transfer@326594a710391a56c58b15ff9146bbe283e6c788

go-sectorbuilder@51775363aa1865e6c3586b939b8d9b3de76b9bb5

go-statemachine@df9b130df3704298a9f19b3f95f190003fefe168

go-statestore@2ee326dbc6d74138893722f842be90e350f5bb23

go-hamt-ipld@af919077d5ae2a5d579c21e1f1c24a345c710a2c

go-ipld-cbor@f88d4ac9d3eb5e6ef7f77f39aead1ebd1ef3c6f7

cbor-gen@c568d328ad9dc887b5103a9dcb0b3645224c8c1f

For the verification, we examined the Git following revisions:

lotus@19d457ae5b1e6583089239852c962acba034a270

go-fil-markets@b4a5c7e9bb95d13ce2aad1c199cdb451112d7835

go-ipld-prime@6e6625bd5fc59f2634575b258da69d8e4aaf1716

go-fil-commcid@d41df56b4f6a934316028e4d4b93fb220674801d

go-padreader@9c5eb1faedb57c6f25b82f992d4e742c94d5086d

go-graphsync@1bdc5585248c9c77b82473ee2d05a4cd6e25db19

go-address@f2023ef3f5bbc513599a3fbf19c4770485146a07

go-amt-ipld@b273a4b34be898897cd272d6a6a118737cc2d749

go-bitfield@fe2c1862e8169d3020b8749340d0d1a275280ae9

go-cbor-util@d0bbec7bfcc45e593be8195a12352563355d2427

go-crypto@effae4ea9f030bfb05c3caaa42eb25bba317d5b7

go-data-transfer@79b3fbd7bdf9a0bf61a2b00f3a5e2196bd5f0e18

go-sectorbuilder@51775363aa1865e6c3586b939b8d9b3de76b9bb5

go-statemachine@aaed5359be39d589fbe7a9f24c4193fd434c5021

go-statestore@8a2d9d6dbd5b8b9a48609d23208b018eb4404e13

go-hamt-ipld@d1f554ae2626245c4ac5b5b698f426d5cfa400c4

go-ipld-cbor@f689d2bb3874cf3fafb71721cafb2c945234e781

cbor-gen@0b9f6c5fb1636544f94f5087817df99699de49ba

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation was available to the review team:

- Filecoin Specification: https://beta.spec.filecoin.io/*
- Implementation Architecture: https://docs.lotu.sh/en+arch
- Implementation Documentation: https://docs.lotu.sh
- Least Authority - Security Audit Hackmd: https://hackmd.io/WBzPQSinSsehUJbkJ-pWoq

*The Filecoin Specification was used as an aid, however, at the time of the review it was incomplete. Section 1.1 Spec Status indicates which sections of the specification are stable, incomplete, incorrect, or a work in progress (WIP).

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;

- Common and case-specific implementation errors;
- Vulnerabilities within individual components as well as secure interaction between the network components;
- Securely handling large volumes of network traffic;
- Adversarial actions and other potential attacks on the network;
- Protection against malicious attacks and other methods of exploitation;
- Resistance to Denial of Service (DoS) and similar attacks;
- Key management implementation, including the secure key storage and proper management of encryption and signing keys;
- Storing assets securely;
- Vulnerabilities within the implementation and potential for loss of funds handled by the implementation;
- Any attack that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;
- Exposure of any critical information during user interactions with the blockchain and any external libraries;
- Networking and communication with external data;
- Data privacy, data leaking, and information integrity;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Lotus, an implementation of Filecoin, is a blockchain-based distributed data storage network where the data is stored off-chain and all of the transactions that occur in a storage cycle are verifiable on-chain. Other approaches to blockchain-backed storage have created centralized sources of truth or stored the data directly in the blocks on the chain. In contrast, Filecoin only stores the proof that a transaction occurred, in addition to a check for what that data is on-chain. Filecoin uses Storage Power Consensus, the weighted total of a node's storage deals, duration, and sector sizes, to determine who mines new blocks and elects new leaders. This creates an incentive alignment that is favorable to clients wishing to store data because storage miners, the nodes that are storing data and helping mine blocks on-chain, are seeking to store as much data as quickly as they can in order to increase their rewards and voting power. To support these incentives, Filecoin has created a market around storage where miners can offer deals at their own declared prices.

In Filecoin, when a node wants to retrieve the data they have stored on-chain, a node submits a `Piece` retrieval request. Since `Piece` retrieval is off-chain, it is not backed by the same security guarantees of the Filecoin blockchain, such as sending or receiving FIL or pushing a `Piece` to the network. Filecoin uses a voucher system combined with payment channels to facilitate rapid and voucher-backed retrieval of files from their respective storage miners. This system introduces some trust at the benefit of vastly increased performance in reading the data. However, because this system is off-chain and fairly closely coupled to the payment channel implementation, we recommend that Filecoin continuously investigate and review the off-chain retrieval process for issues and vulnerabilities.

As is the case with any project presenting new concepts and technology, this will inevitably result in challenges, unknown risks, and subsequent lessons for both the project and the industry, as Filecoin experiences production-specific issues following the mainnet launch. However, the Protocol Labs team has made reasonable trade-offs and demonstrates excellent engineering and a well thought out design behind the product. This is particularly evident through the Protocol Labs team's bottom-up approach, in which they have broken down the Filecoin protocol into a number of smaller modules, with limited

exposure and responsibility. As a result, they have kept security-critical components in a limited area of operation, thus minimizing the attack surface to potential vulnerabilities. Lotus, the wrapper that combines all of them into a single package, delivers the full experience to the end user.

Audit Strategy and Approach

Our team followed an intuitive analysis approach and manually reviewed the code, along with utilizing a variety of tools and strategies for the duration of the audit. We conducted static analysis using open source tools, including SonarQube and gosec, which revealed no major issues.

Fuzz Testing

In addition, we conducted extensive fuzz testing and focused our fuzzing efforts on low level data processing libraries and functions (most notably `go-ipld-prime`) as well as network payloads (`go-graphsync` and `go-fil-markets` in particular) with the intention of identifying messages that an attacker might purposefully create, resulting in security issues. Through that process, we discovered several issues in the `go-ipld-prime` module (Issue C; Issue D; Issue E; Issue F). We also identified crashing metadata inputs in the `go-graphsync` implementation (Issue B). Despite our efforts, we strongly recommend that additional fuzz testing be conducted (Suggestion 4).

It is important to note that fuzz testing presents some difficulty in assessing and determining the severity of an issue. Since Filecoin is composed of a number of smaller modules, which call each other at different points and locations, a fuzz input might be a non-issue in one module but may potentially be a high severity bug in a different location and call stack. Our team's fuzz test setup and known crashing inputs can be provided to the Protocol Labs team in order to facilitate any future fuzz testing that is carried out independently by their team.

System Design

Our team found Lotus to be a cohesive and well-defined implementation of the larger Filecoin system. While the overall design of Filecoin is excellent and demonstrates strong considerations for security, we identified several security critical areas of the system that warrant further investigation. As a result, we recommend further review and enhanced protection for end users in the following areas noted in this section.

Payment Channels

Filecoin makes a distinction between on-chain and off-chain transactions. A `Piece` is "an object that represents a whole or part of a `File`" and pushing a `Piece` onto the system is an on-chain transaction, while retrieving a `Piece` is an off-chain transaction. Filecoin achieves off-chain retrieval actions with vouchers: a requesting node will create a voucher that contains their wallet address, the `PieceCID` that they want, a small fee for handling the `Piece` retrieval called an `UnsealPrice`, and several other pieces of data. However, since the node is simply agreeing to pay the price and not actually processing the payment, the retrieval off-chain actions require implicit trust and it is possible that a node defaults on a payment

To address this dependency, we recommend that unpaid vouchers be tracked and possibly penalized at the network layer (Suggestion 2) in the event that a node defaults on a payment. If a node sends a retrieval request with a corresponding voucher and does not remit payment, they should not be served until the node is paid in full for their previous request.

DoS Attacks

Our team did not find measures in place for rate limiting or throttling at the node level, which is considered to be an added layer of protection against DoS attacks. Although complete protection against DoS attacks of sufficient power is impossible, it is possible to decrease the overall effectiveness of an attack while simultaneously increasing the power and bandwidth needed to effectively harm or slow a target node. We recommend adding a general rate limiter on requests as an extra layer of protection for each node (Suggestion 1).

Furthermore, the bandwidth-optimized `Piece` retrieval in the Filecoin system is particularly susceptible to attacks of this nature, and that any possible mitigation should be employed to weaken or discourage such attack vectors.

Peer Scoring

Lotus, along with all other Filecoin full node implementations, utilizes the Gossipsub protocol. Gossipsub allows for peer scores to be updated by an arbitrary function. While the base Gossipsub layer utilizes the standard node penalties and will slash bad networking behavior, we recommend also penalizing known bad behavior by nodes at the Filecoin layer. For example, in Issue A, repeated transmission of syntactically valid requests with unacceptable deal parameters could be recognized and penalized if repeated past any given amount of times (Issue A). This would help prevent DoS attacks while still allowing for a margin of error for the majority of users.

Code Quality + Documentation

As previously noted, Lotus and the Filecoin system have been engineered with a bottom-up approach. While the code base is substantial in size and comprehensive in detail, thus increasing the learning curve for the system, it becomes significantly easier to navigate once a basic understanding is reached for each of the modules and how they relate.

Navigation of the code is aided by the code being very well organized, with a clear and concise separation of concerns. Each module has a well-defined interface that is strictly enforced, also helping to reduce the risk for the introduction of errors. The implementation details are mostly kept in an `` `impl` `` package within each module, making it easier to become familiar with the code within each module and review for potential security issues. The practice of loose coupling between modules is good for code reuse and adaptability and more efficiently allows reviewers to understand the system. In addition, the Go idiomatic code and widespread use of encryption adheres to secure programming best practices.

Test coverage is extensive and frequent. Our team did not identify any security critical areas that were not tested with happy path and error handling tests. The Protocol Labs team used a combination of table tests and declarative tests throughout the project, with a particular focus on important parts of the codebase, rather than setting an arbitrary test coverage number. This demonstrates a strong consideration and thorough planning in order to optimize the security of the implementation.

We commend the Protocol Labs team for providing thorough and comprehensive project documentation, providing broad insight into all aspects of the system design and implementation. The consistency between the project documentation and the coded implementation is notable and allowed our team to effectively check the correctness of the implementation and understand the system architecture. Code comments provide clear detail and insight into the intended behavior and functionally, which is very helpful in both familiarizing reviewers with the code in the implementation. In addition, while the Filecoin Specification is currently incomplete, it is thoroughly defined and clearly specifies the status of the sections within the specification and whether they are stable or a work in progress, which minimizes the risk of confusion. While we do not consider the incompleteness of the specification to be a security issue,

we recommend that further updates to the system that correspond with updates to the specification be followed with regular reviews and security audits.

Out of Scope Dependencies

Specs-Actors

The specs-actors repository was out of scope for our audit, however, it is a security critical piece of the system. It is defined as "the specification of the Filecoin builtin actors, in the form of executable code" and "a companion to the rest of the Filecoin Specification, but also directly usable by Go implementations of Filecoin", including Lotus. This is similar to the way in which the Ethereum 2.0 Beacon Chain is specified. One important distinction, however, is that the specs-actors code is considerably more complex than that of the Ethereum 2.0 specification. With increased code complexity, the chance for hidden assumptions and edge cases also increases, which results in a larger surface area for potential bugs and security vulnerabilities to appear. As a result, we recommend an in-depth audit of specs-actors and its relationship with the Filecoin network and chain in order to conduct an in-depth examination of those interactions.

Utility Libraries

Lotus utilizes and depends on several small utility libraries that were not included in the scope of this audit. While we are unable to report on the security of these dependencies, they are internal libraries developed and maintained by the Protocol Labs team. Furthermore, these libraries are sufficiently covered by tests in the packages that call them. As a result, there are minimal security concerns around potential vulnerabilities and can thus be categorized as an implementation detail in the system.

Specific Issues

We list the issues found in the code, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: requestvalidation: Piece Requests Access Disk Before Checking Validity of Request	Resolved
Issue B: message: FromNet Inputs Produce an Index Out of Range Error	Resolved
Issue C: dagcbor: Input toUnmarshal Function Causes Panic	Resolved
Issue D: dagcbor: Parsing Adversarially Chosen Data Causes Out-of-Bounds Slice Read	Resolved
Issue E: dagcbor: Parsing Adversarially Chosen Data Crashes Node Due to Memory Exhaustion	Resolved
Issue F: dagjson.Encoder: Lack of Float Support in refmt Causes a Crash	Resolved
Suggestion 1: Implement Per-Node Rate Limiting	Unresolved

Suggestion 2: Penalize Know Bad Actor Behavior at the Network Level	Unresolved
Suggestion 3: metadata: DecodeMetadata Accepts Empty CIDs, but Encoder Does Not	Unresolved
Suggestion 4: Conduct Additional Fuzz Testing	Unresolved

Issue A: requestvalidation: Piece Requests Access Disk Before Checking Validity of Request

Location

https://github.com/LeastAuthority/go-fil-markets/pull/1

Synopsis

An attacker can generate and send a valid `Piece` request to a target `StorageMiner`, specifically, any `RetrievalMarket` provider that triggers an unnecessary disk access before being validated. This disk access can be exploited to consume a node's resources.

Impact

We consider this a high impact issue, which has the potential to flood any node in the system with valid requests that would immediately return at no expense to the attacker, with the exception of the initial storage cost for the `Piece`.

Preconditions

The `StorageMiner` must store a `Piece` for the attacker and the attacker must format a request for a `Piece` in such a way that it is valid except for the `UnsealPrice`, `PaymentIntervalIncrease`, `PaymentInterval`, or `PricePerByte`. This still causes the disk to be accessed but the payment terms to be rejected.

Feasibility

This attack can be carried out by a single actor with a Filecoin node and a fair amount of knowledge of the Filecoin specification and implementations.

Technical Details

The attacker stores a `Piece` with a given `StorageMiner`. As soon as the `Piece` is stored, the attacker can begin crafting a request that would be out of the range of the `StorageMiner`'s accepted payment parameters (in this instance, the configurable parameters are `deal.PricePerByte`, `deal.PaymentInterval`, `deal.PaymentIntervalIncrease`, or `deal.UnsealPrice`) and start flooding the victim with requests for that `Piece`. When the `RetrievalMarket` provider goes to handle the retrieval request, they check that they have the `Piece` in storage, which then triggers a disk read and loads the `Piece`. The `RetrievalMarket` provider then validates the price parameters, which returns an error and stops the request. This is significant because it means it will not cost the attacker anything to send requests, as `vouchers` are not processed if the request is denied for being out of bounds, however, they are able to incur unnecessary and non-trivial disk usage to check. An attacker can flood a victim `StorageMiner` with these requests and cause delays in their disk reads, slow down request handling for other users, and potentially stall them out of mining.

Remediation

We propose validating the request before the disk access so that the attacker cannot force the disk access without being required to pay.

Status

The Protocol Labs team implemented the suggested remediation of checking the validity of the payment parameters prior to checking the disk for the `Piece`.

Verification

Resolved.

Issue B: message: FromNet Inputs Produce an Index Out of Range Error

Location

https://github.com/LeastAuthority/go-graphsync/issues/2

https://github.com/LeastAuthority/go-graphsync/blob/master/message/pb/message.pb.go

Synopsis

We discovered crashing inputs while fuzzing the `FromNet` function in `go-graphsync`.

Impact

We consider this a low impact issue. An attacker could send this payload to a victim node and crash the thread handling the request. However, once the thread is crashed, the attacker has no more control or input into the system and the connection is terminated.

Preconditions

The attacker must be capable of making `go-graphsync` requests with custom payloads. They must also have a way to generate this input or knowledge of this crashing input.

Feasibility

This attack can be carried out by a single actor with a `go-graphsync` capable node and a fair amount of knowledge of the Filecoin protocol and implementations.

Technical Details

During fuzz testing, we discovered crashing inputs to the `FromNet` and `ToNet` functions in `go-graphsync`. An attacker would craft a `go-graphsync` request with this payload as the metadata extension:

```
   "$\x1a \x8000\x1a\x16002\xf4\xff\xff\xff\xff\xff\xff\xff\xff" +
"00000000000000000"
```

During processing this input, the thread panics and produced the following output:

```
`panic: runtime error: index out of range [-9223372036854775802]`
```

Mitigation

A proper remediation is beyond the scope, since it is an issue in the gogo-protobuf code generator. As a mitigation, we recommend changing the generated code to ignore unknown fields in all messages and

submessages. To do this, return early in the `default` cases in all `switch` statements on variables of type `fieldNum`. In our tests, this eliminates this class of crashes.

Status

The Protocol Labs team is now using Google's protobuf generator instead of the gogo-protobuf code generator. In addition, they have added a regression test for the crashing input discovered during the audit, in order to protect against any potential issues in future releases.

Verification

Resolved.

Issue C: dagcbor: Input to Unmarshal Function Causes Panic

Location

https://github.com/LeastAuthority/go-ipld-prime/issues/4

Synopsis

While fuzz testing the `DecodeMetadata` function in `go-graphsync`, we discovered an input that crashes the `goroutine` processing it.

Impact

We consider this a low to moderate impact issue. This crashing input could be sent to nodes to eat up processing time and force resource usage. However, the damage is limited since the connection is dropped once the node crashes the `goroutine`.

Preconditions

The `lotus node` must be running a vulnerable version of `go-graphsync`.

Feasibility

This attack requires the attacker to run a `go-graphsync` node and a familiarity with the protocols, as well as knowledge of the crashing input or a way to generate it themselves.

Technical Details

An attacker sends a `go-graphsync` request to a target `go-graphsync` node with a specific payload containing the raw input of

```
"\xbf\u007f\xff\x8c\xbf\u007f\xff\x8c\xbf\u007f\xff\x8c\xbf\u007f\xff\x8c\xbf\
u007f\xff\xbb" + "00000000”
```

When the node receives the request, they will attempt to decode the extension metadata and, in doing so, run across this input which will crash the `DecodeMetadata` function and the entirety of the `goroutine` handling that request.

Remediation

This bug can be fixed with a simple length check for the token in the `dagcbor unmarshal` function.

Status

The Protocol Labs team implemented a gas budgeting system in the `Unmarshal` function that limits the amount of memory that can be consumed while processing a message. This prohibits the `Unmarshal`

function from crashing the `goroutine` that is processing the message by early returning if it meets the allocated memory budget.

Verification

Resolved.

Issue D: dagcbor: Parsing Adversarially Chosen Data Causes Out-of-Bounds Slice Read

Location

https://github.com/LeastAuthority/go-ipld-prime/issues/7

Synopsis

We discovered an input to the `dagcbor unmarshal` function that produces an index out of bounds panic.

Impact

We consider this a low to moderate impact issue. This crashing input may be sent to nodes to consume processing time and force resource usage. However, the damage is limited since the `goroutine` handling the request recovers from the panic.

Preconditions

The target node must be using a vulnerable dependency version of `go-ipld-prime` and running a `go-graphsync` or equivalent network-capable node. The attacker must be connected to the target and have an active `go-graphsync` session. This is the case for any data transfer connection that two peers have.

Technical Details

The first element of `tk.Bytes` is accessed without checking whether the slice is empty. This results in an out of bounds panic.

Remediation

Before accessing an element of a slice, make sure that the slice is long enough.

Status

The Protocol Labs team added a simple length check that prevents the out of bounds access attempt and, as a result, the panic.

Verification

Resolved.

Issue E: dagcbor: Parsing Adversarially Chosen Data Crashes Node Due to Memory Exhaustion

Location

https://github.com/LeastAuthority/go-ipld-prime/issues/6

https://github.com/LeastAuthority/go-ipld-prime/tree/master/node/basic

https://github.com/LeastAuthority/go-ipld-prime/tree/master/codec/dagcbor

Synopsis

During fuzz testing we discovered inputs that cause an out of memory error when decoded using `dagcbor`.

Impact

The triggering of the issue unrecoverably crashes the `lotus node` due to running out of memory. This may leave the persistent database in a corrupted state, so simply restarting it may not be possible.

Preconditions

The attacker needs to be in an active `go-graphsync` exchange network with the node under attack.

Feasibility

Since `go-graphsync` is a core part of Filecoin and it is used to transfer data between miners and clients, it is relatively easy for a client to crash a miner.

Technical Details

The central issue is that the `go-ipld-prime` performs no sanity checks of the encoded data and imposes no boundaries on allocated resources. Specifically, the encoding of arrays in CBOR may include an element count. The attacker can create a CBOR-object that contains an array with a very high element count and the `refmt` library will attempt to allocate the corresponding memory.

Remediation

We suggest enforcing a limit of memory that the CBOR parser is allowed to allocate. The limit may be hardcoded or specified by the calling function. We suggest a default limit of 64MiB, following the example of the JavaScript package ipld-dag-cbor. In general, we recommend treating the CBOR data as untrusted user input.

Status

The Protocol Labs team implemented a gas budgeting system in the `Unmarshal` function that limits the amount of memory that can be consumed while processing a message. This prohibits the `Unmarshal` function from crashing the `goroutine` that is processing the message by early returning if it meets the allocated memory budget.

Verification

Resolved.

Issue F: dagjson.Encoder: Lack of Float Support in refmt Causes a Crash

Location

https://github.com/LeastAuthority/go-ipld-prime/issues/5

https://github.com/polydawn/refmt/blob/3d65705ee9f12dc0dfcc0dc6cf9666e97b93f339/json/jsonEncoder.go#L211

Synopsis

During fuzz testing we discovered inputs, which can be decoded but then crash the encoder when re-encoding. It appears that in this particular case, it is of a number that gets parsed as a float which is not supported by `refmt`'s encoder.

Impact

This issue has the potential to crash a `lotus` node unless it is recovered from by an out of scope system unknown to our team.

Preconditions

The attacker needs to be in an active `go-graphsync` exchange network with the node under attack.

Feasibility

Since `go-graphsync` is a core part of Filecoin and it is used to transfer data between miners and clients, it is relatively easy for a client to crash a miner.

Technical Details

`Go-ipld-prime` depends on the `refmt` module for object serialization. `Refmt`'s JSON encoder currently has limited support for JSON primitives, which results in a panic when it encounters an unsupported input type. If a node attempts to re-encode this input (which it can decode without error), it will crash.

Remediation

Adding a recovery statement where `refmt` is being used would allow the program to regain control after experiencing a panicking call to the encoder.

Status

The Protocol Labs team implemented float support to the underlying `refmt` library that was panicking when it attempted to encode the input, thus resolving this issue.

Verification

Resolved.

Suggestions

Suggestion 1: Implement Per-Node Rate Limiting

Synopsis

Although complete protection against DoS attacks of sufficient power is impossible, it is possible to decrease the overall effectiveness of an attack while simultaneously increasing the power and bandwidth needed to effectively harm or slow a target node.

Configurable, per-node request rate limiting would mitigate a large class of DoS attacks and allow nodes to still ultimately control their nodes.

Mitigation

Add a rate limiter for requests from unique nodes that is configurable to the end user.

Status

The Protocol Labs team responded that they acknowledge the validity of this suggestion and stated that they currently have open issues in their repositories for adding fine-grained rate limiting controls to nodes. They have also noted that they do not plan on addressing this suggestion in the shorter term and it remains unresolved at the time of this verification.

Verification

Unresolved.

Suggestion 2: Penalize Known Bad Actor Behavior at the Network Level

Location

`ScoreKeeper` hook that is available on libp2p implementations.

Synopsis

Nodes behaving in a way that is not constructive waste the resources of honest nodes. In order to make such behavior less draining on resources, peer scoring can be used to penalize bad behavior and provide fewer resources to nodes displaying it. Such a system is already present in Lotus (`PeerScorer` in dtypes) and could be accessed here and loosely coupled through an interface that implements `Get` and `Update`.

Mitigation

Lotus already has a `ScoreKeeper` interface (`Get` and `Update`) defined at `` `lotus/node/modules/dtypes` `` for external modification of node peer scores. We recommend using this interface for penalizing nodes that show bad behavior at the protocol level (`go-fil-markets`, `go-graphsync`, etc.) and not only the GossipSub network level.

Status

The Protocol Labs team responded that they agree with this suggestion and will investigate it further in the future. At the time of this verification, this suggestion remains unresolved.

Verification

Unresolved.

Suggestion 3: metadata: DecodeMetadata Accepts Empty CIDs, but Encoder Does Not

Location

https://github.com/LeastAuthority/go-graphsync/issues/1

Synopsis

The metadata decoder in `go-graphsync` accepts empty CIDs during decoding, even if these values are typically invalid. Notably, the encoding function will fail with an error when encoding a data structure that contains empty CIDs. If inputs are accepted, then all edge cases must also be handled. In this case, all code needs to handle empty CIDs (i.e. check if they are cid.Undef). This check is a form of input validation.

Mitigation

If there are no circumstances in which unvalidated inputs should be processed, we recommend the validation occur in the decode function (which is in `ipld_cbor_gen`). In circumstances where validation is not desired, a mechanism should be used where reading the calling code identifies when unvalidated decoding is used. Performing the validation should be the default.

There are two possible options to do this: using different decode functions (e.g. `Decode` and `DecodeUnvalidated`) or using options for a single decode function (e.g. using functional options: `Decode(cid.AllowEmptyCID)`).

Status

The Protocol Labs responded that they acknowledge that empty CID handling is an issue across several subsystems and state that they intend to update them for consistency. At the time of this verification, this suggestion remains unresolved.

Verification

Unresolved.

Suggestion 4: Conduct Additional Fuzz Testing

Location

https://github.com/filecoin-project/go-bitfield

https://github.com/filecoin-project/go-cbor-util

https://github.com/whyrusleeping/cbor-gen

https://github.com/filecoin-project/lotus/tree/master/paychmgr

https://github.com/filecoin-project/lotus/tree/master/chain/vm

https://github.com/filecoin-project/go-data-transfer

https://github.com/LeastAuthority/go-ipld-prime/

Synopsis

In particular, `go-bitfield`, `go-cbor-util`, `cbor-gen`, `lotus/paychmgr`, `go-ipld-prime`, and `go-data-transfer` warrant further testing and investigation:

- `go-bitfield` would affect Lotus miners if crashers were discovered.
- `go-cbor-util` and `cbor-gen` packages, there are functions which process binary data from the network, thus making them extremely interesting targets from a fuzzing and attack vector perspective. Fuzzing these critical serialization functions would exercise novel code paths where processes could crash.
- `go-data-transfer` since it interacts heavily with both the file system and the network layers.
- `lotus/paychmgr` since it wraps their Payment Channel implementation.
- `lotus/chain/vm` since it wraps their virtual machine and specs-actors implementation.
- `go-ipld-prime` since it has been a source of two issues so far, has a lot of control flow based on user input and would benefit from additional scrutiny and fuzzing.

Mitigation

Our team recommends this additional fuzz testing be performed and can share our fuzz testing tools and methodology with the Protocol Labs team.

Status

The Protocol Labs team responded that they intend to incorporate additional fuzz testing into upcoming audits in 2021.

Verification

Unresolved.

Recommendations

We recommend that the unresolved *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

We recommend additional review of the areas noted above, along with fuzz testing of core Filecoin components since these components are input-heavy and process external data. We also recommend an in-depth audit of specs-actors and its relationship with the Filecoin network and chain.

We commend the Protocol Labs team for well organized code and thorough and comprehensive project documentation.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create

an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.