Least Authority
PRIVACY MATTERS

Venus

Security Audit Report

Filecoin Foundation

Final Report Version: 29 June 2021

Table of Contents

Overview

Background

Filecoin Foundation requested that Least Authority perform a security audit of Venus, an implementation of the Filecoin Distributed Storage Network written in Go.

Filecoin is a decentralized storage network that transforms unused cloud storage into an algorithmic market in which miners and clients are incentivized to participate.

Project Dates

- **March 8 - April 16**: Code review *(Completed)*
- **April 22**: Delivery of Initial Audit Report *(Completed)*
- **June 24 - June 28**: Verification Review *(Completed)*
- **June 29**: Final Audit Report delivered *(Completed)*

Review Team

- Bryan White, Security Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Rai Yang, Security Researcher and Engineer
- Steve Thakur, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Venus 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:

- Venus: https://github.com/filecoin-project/venus/tree/v0.9.1

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

`f7b073a29b2fb181b8af28e9fe9ca225e1c085bb`

For the verification, we examined the Git revision:

`8ea74190ba237214e1e2c1436362b9b030f936f3`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/Filecoin-Venus

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

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Protocol Specification: https://spec.filecoin.io/*
- Filecoin Documentation: https://docs.filecoin.io/get-started/#filecoin-implementations
- `README.md`: https://github.com/filecoin-project/venus/blob/master/README.md
- `CODEWALK.md`: https://github.com/filecoin-project/venus/blob/master/CODEWALK.md
- Venus Documentation Repository (still in progress): https://github.com/filecoin-project/venus-docs

In addition, this audit report references the following documents:

- Blog post, "The Scrypt Parameters": https://blog.filippo.io/the-scrypt-parameters/
- P. Maymounkov, D. Mazières, 2002, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric." *In: Druschel P., Kaashoek F., Rowstron A. (eds) Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science, vol 2429*. Springer, Berlin, Heidelberg. [MM02]

*The Protocol Specification can be used as an aid, however, it is 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

Filecoin is a peer-to-peer network made up of Filecoin client nodes (clients) that participate in a distributed file storage network with Filecoin miner nodes (miners), which execute data storage and retrieval calls and provide data integrity and security. The network leverages a native protocol token, Filecoin (FIL), to create an incentive structure and facilitate the negotiation of data storage and retrieval services between clients and miners. Miners earn FIL by providing data storage and retrieval while clients pay miners for data storage or distribution and retrieval. Venus, a Go implementation of the Filecoin Distributed Storage Network, was the core focus of our security review.

System Design

The Venus system design is well defined and adheres to the overall design of the Filecoin system, which demonstrates strong considerations for security. However, the Venus coded implementation is considerably large and characterized by a high level of complexity. In particular, the interaction of multiple critical components is further complicated by those components running submodules, which also perform critical functions and intricate interactions. While our team did not identify any critical security vulnerabilities in the design of the system, we did identify several issues in the Venus implementation. Systems characterized by a high degree of complexity are exposed to a larger attack surface, which may result in hidden errors leading to critical security issues. We recommend that the Venus team continue to pursue opportunities to reduce the complexity of the implementation if and where possible. In addition, we suggest that regular security audits of the Venus implementation be conducted by different teams to mitigate against potential vulnerabilities. This is particularly pertinent at junctures where the existing system design and the implementation undergo changes that introduce updates or new functionality.

Incentive Mechanism

The Venus incentive mechanism functions such that miners are compensated in FIL to store data. Miners are required to prove that a unique copy of the data is stored by Proof of Replication and prove continuously that the data is being stored using a Proof of Spacetime. Miners are also compensated for executing storage and retrieval tasks. In turn, the proofs will be verified by clients and miners will be penalized for providing the fault proof or committing false consensus faults. We did not identify any issues in the implementation of the incentive mechanism.

Actor Model

The Filecoin Virtual Machine (VM) uses an actor model to manage state, in which eleven types of built-in actors maintain the state of the system in the state tree. The actors method can be invoked by messages sent by other actors or by the system itself. Message invocation will drive the state change of the system. In the case of Venus and the Filecoin system at large, actors are the equivalent of smart contracts in other ecosystems. We did not identify any issues in the implementation of the actor model to manage state.

Distributed Hash Table (DHT)

The Filecoin Network utilizes a DHT-based routing design (i.e. [MM02]), which efficiently allows a node's IP address to be located by node ID. However, this type of design potentially allows miner nodes to become targets of Distributed Denial of Service (DDoS) attacks, severing a miner from the network and inhibiting its responsiveness to challenges, which results in the miner being penalized. While this may have an impact on a miner's uptime, this type of attack is common to all DHT-based systems. However, the Filecoin protocol was designed to be resilient against such attacks and, as a result, the DHT-based routing design does not pose a serious threat to the overall security and stability of the network.

Trusted Setup

The Venus implementation's Common Reference String (CRS) requires a trusted setup that utilizes a somewhat centralized MPC ceremony, with limited participants and transparency. While this system component is out of scope, it is worth noting that it introduces the possibility of collusion and compromising of the proof system. In addition, the system implements a Groth16 zk-SNARK proof, creating a one-time and non-updateable CRS. We recommend that a security audit of the trusted setup be performed by an independent team, in order to explore ways to protect the network and users from compromise of the proof system (Suggestion 9).

Code Quality

The Venus implementation is well organized in some parts of the code base. However, some sections of the coded implementation make use of an inconsistent naming convention, which results in difficulty reading and understanding the code. We recommend adhering to a naming convention for variables and functions that is descriptive and accurate, and that is consistent with generally accepted best practices (Suggestion 4).

Tests

Several of the packages in the Venus repository contain unit and integration tests. However, a significant proportion of the packages are missing tests. A robust test suite helps to identify errors and bugs that may lead to potential vulnerabilities. As a result, we recommend increasing test coverage to include all packages, with tests for success and failure cases to account for edge cases, unexpected, and unintended behavior (Suggestion 3).

Fuzz Testing

As a supplement to our manual review of the code, our team identified prospective fuzz testing targets and identified several data structures which implement a CBOR codec from partially generated code via `whyrusleeping/cbor-gen`. We prioritized the prospective targets list based on potential severity and implemented fuzz tests on the following targets:

- `tools/conformance/chaos/state.go`: `State`
- `pkg/chainsync/exchange/protocol.go`:
  - `Request`
  - `BSTipset`
- `pkg/types/signed_message.go`: `SignedMessage`
- `pkg/crypto/keyinfo.go`: `KeyInfo`

The results from our fuzz testing efforts included only false positives.

Documentation

The Venus implementation is supported by thorough and comprehensive documentation, including a CODEWALK.md explaining the intended functionality of the code and the Filecoin Specification, which provides a comprehensive overview of the system. 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. We commend the Filecoin team for being rigorous in their efforts to provide clear, concise, detailed, and up-to-date documentation.

Code Comments

The Venus code base contains minimal code comments describing the intended functionality of the system. Furthermore, the exemption of missing comments is enabled in the linter. Code comments help increase visibility into the intended functionality of the code and facilitate a more thorough understanding of the system, which is beneficial for users, security researchers, and the overall security of the system. We recommend disabling the exemption of lacking comments in the linter and increasing code comment coverage, which reduces the opportunity for errors and misunderstanding the intended functionality of the code (Suggestion 1).

Scope

The scope of the security audit was sufficient in that it encompassed the entire implementation, including all security critical components of the system. While the dependencies used by the Venus implementation were not in scope, the use of dependencies is mostly limited to standard libraries that are both well audited and maintained. For example, Venus makes use of the `specs-actors` dependency, which performs a core functionality of the implementation. While `specs-actors` was not in scope, it has recently undergone an independent security review.

However, we found that Venus utilizes the deprecated packages `go-multiaddr-net` and `go-libp2p-crypto`. We recommend replacing deprecated packages with the replacement packages provided by the authors (Suggestion 8).

Finally, as noted previously (see Trusted Setup), we recommend a follow up audit of the trusted setup, which was out of scope for this review (Suggestion 9).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, 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: Weak Scrypt Due to Low Parameters	Resolved
Issue B: Node Initialization Loads a Validator for Namespace "v" Accepting Every Message	Resolved
Issue C: Locking a Wallet Requires a Password	Resolved
Issue D: Keys Lack Adequate Protection	Partially Resolved
Issue E: Incorrect Logic for Comparing Sync Target's Parent Weight and Missing Parent Check in IsNeibor Function	Resolved
Issue F: Program Panics if Length of Returned Messages is Less Than Length of Tipset Segments Requested	Partially Resolved
Issue G: Target Queue Size Exceeds the Preset Maximum Size	Resolved
Suggestion 1: Increase Code Comments and Disable Linting Exemption	Partially Resolved
Suggestion 2: Reduce Syncing Time	Unresolved
Suggestion 3: Increase Test Coverage	Partially Resolved
Suggestion 4: Correct Inaccurate and Misleading Names	Resolved
Suggestion 5: Remove Duplicate Code	Resolved

Suggestion 6: Use 32 Random Bytes as HMAC JSON Web Token Secret	Resolved
Suggestion 7: Check That Linting Works as Expected	Resolved
Suggestion 8: Switch Away From Deprecated Packages	Resolved
Suggestion 9: Conduct an Audit of the Trusted Setup	Unresolved

Issue A: Weak Scrypt Due to Low Parameters

Location

`pkg/config/config.go#L19-L22`

Synopsis

The `scrypt` function is a memory hard function designed to securely derive keys from passwords. The caller can specify parameters, such as iterations and required memory. However, the parameters chosen by the client are too low.

Impact

Setting low `scrypt` parameters more easily facilitates an attacker's ability to successfully guess a low-entropy password. This would enable an attacker to unlock a user's wallet, which can result in the loss of user funds.

Preconditions

A low-entropy password (which is common) and physical access to the encrypted wallet of the target.

Feasibility

This attack requires significant computational resources but is technically feasible. This attack is economically feasible if the profits gained from compromising the wallet surpass the costs of the computational resources.

Technical Details

An established standard for `scrypt` parameters does not exist and the `scrypt` parameters in Venus are implemented as suggested in an article on scrypt parameters, written in 2017. The Venus implementation makes use of the scrypt parameter recommendations for interactive logins but uses scrypt for computing file encryption keys. These lower interactive login parameter settings produce much higher performance than what is required for unlocking a wallet, resulting in a much lower cost of computing hashes and decreasing the time needed to bruteforce the password.

Remediation

We recommend using a minimum of `scrypt` parameter `N = 1 << 21`.

Status

The Venus team has updated the `scrypt` N parameter to 1 << 21. The lower value 1 << 15 is still used in tests, in order to maintain short test execution times.

Verification

Resolved.

Issue B: Node Initialization Loads a Validator for Namespace `"v"` Accepting Every Message

Location

[`submodule/network/network_submodule.go#L133`]()

Synopsis

The network submodule uses a DHT validator that allows all records in the "v" namespace. While the purpose of namespace is unclear, it does constitute a validation bypass.

Impact

Messages in the "v" namespace are not being adequately validated, which allows unvalidated messages to enter the system. This may interfere with the correct processing of valid messages.

Technical Details

Networked applications should first check incoming messages for validity. The function that does this is called a validator. In this instance, a validator that allows all messages is explicitly loaded, leading to a bypass of validation.

Remediation

We recommend removing the `blankValidator` from the DHT initialization.

Status

The Venus team has [removed]() the validator from the initialization.

Verification

Resolved.

Issue C: Locking a Wallet Requires a Password

Location

[`pkg/wallet/dsbackend.go#L241-L250`]()

Synopsis

The wallet requires the input of the password in order to lock and encrypt the data, slowing the user's response to real-world threats, and increasing risk of unauthorized access. When needed, placing a wallet in a locked state must always be easy and quick, enabling a user to secure their assets. Furthermore, a locked wallet would require that a password is entered before new transactions can be made.

Increased difficulty in locking the wallet inhibits the user from being able to quickly respond to real-world threats to the security of their digital assets. An attack could be as simple as pushing the user away from the computer and using the unlocked wallet.

Impact

An attacker is able to access an unlocked and unencrypted wallet, which could lead to the loss of user funds.

Preconditions

An attacker must separate the unlocked device from the user before the user is able to lock the device.

Remediation

We recommend removing the requirement for a password in order to lock the wallet.

Status

The Venus team has changed the implementation so that it no longer requires entering the password for locking the wallet.

Verification

Resolved.

Issue D: Keys Lack Adequate Protection

Location

`pkg/wallet/dsbackend.go`

Synopsis

Upon locking, the wallet does not delete the Keccak hash of the password from memory, which facilitates efficient password checking. When attempting to unlock the wallet, the password is Keccak-hashed and compared against the stored hash. This operation is very efficient, which is not desirable. Optimally, checking a password should take a long time, in order to make brute-force and dictionary attacks difficult.

The fact that the wallet still contains the Keccak hash in locked states means that an attacker with access to a memory dump of a locked wallet is able to derive the wallet encryption key. If an attacker also has access to the encrypted wallet, this allows them to decrypt the wallet and obtain the secret key.

Additionally, secret keys are never cleared from the memory and may be moved by Go's memory management system or swapped to disk by the operating system, further reducing the security of the secret keys.

Impact

An attacker may get hold of the secret keys of a wallet.

Preconditions

The attacker needs access to the encrypted wallet, in addition to the API or memory of a locked wallet that has been previously unlocked and then locked.

Feasibility

The feasibility varies with the specific setting, but it is to be expected that the attack is possible, yet expensive if the preconditions are fulfilled. The economic feasibility would be determined by the expected reward of an attack, which may vary.

Technical Details

In Venus, the wallet encryption key is derived from the Keccak hash of the passowrd. Locking the wallet only removes the encryption key, but not the Keccak hash, leaving a critical value in memory. Computing the `scrypt` hash of a Keccak hash of a password is not a common practice, which we suspect is implemented to speed up the password checking process. This could be counterproductive because slower password checking mechanisms are a good protection against brute force and dictionary attacks.

Keys are also never overwritten with zeros, and no other effort is made to secure secrets stored in memory (e.g. to prevent Go's memory management from copying memory segments arbitrarily). These may also be written to disk for swap and virtual memory, making the secret values even more accessible.

Given that all of these security issues can not be completely remediated, we recommend implementing both the mitigation and the remediation, as detailed below.

Mitigation

We recommend using `memguard` to protect keys in memory against Go's memory management and the operating system's swap, as well as protect against memory dumps. `Memguard` provides its own memory management, encrypts the memory contents, and sets a non-swap flag.

To mitigate against secrets written to swap, we recommend advising users to disable or encrypt swap on the machines that they use for Venus.

Remediation

We recommend that the implementation does not compute an intermediate hash and only computes the `scrypt` hash of the password. When locking, we recommend overwriting the value and all decrypted secrets with zeros.

Status

The Venus team has implemented a change such that all secrets that are kept in memory throughout the run of the program are stored in `memguard` protected memory. These measures cover the more feasible and likely scenarios of data leakage. However, secret data that is only kept temporarily is often still stored in memory that is managed by the Go runtime, allowing the opportunity for data leakage. While there are some instances where moving secret data from `memguard` managed memory to runtime managed memory can be avoided, we acknowledge that it is often not possible to use `memguard` managed memory when interfacing with external code. In this particular instance, the Venus team has no influence on how allocations are performed externally.

Verification

Partially Resolved.

Issue E: Incorrect Logic for Comparing Sync Target's Parent Weight and Missing Parent Check in `IsNeibor` Function

Location

`chainsync/types/target_tracker.go#L32`

Synopsis

When adding a new sync target in the target queue, the new target is compared with its neighbor target of the same parent weight. If the neighbor target is idle, it is replaced by the new target. The function `IsNeibor` incorrectly checks the equality of the weights of the parents of the two targets, instead of checking that the two targets have the same parents.

Impact

The sync target is merged (replaced) with a new target of the same height, but of a different parent weight or different parents, resulting in an inconsistent chain weight among different clients, potentially delaying or preventing the clients converging into the chain of highest weight or the chain with highest weight among different forks.

Preconditions

A new sync target of the same height from existing targets is found with different parent weight or different parents, and the current target's syncing state is idle.

Feasibility

This is feasible when syncing,the target tipset is in an idle (waiting) state and a new target tipset of the same height is found but with different parents or different parent weight.

Technical Details

In function `IsNeibor`,

```
weightIn := target.Head.ParentWeight()

targetWeight := target.Head.ParentWeight()

if !targetWeight.Equals(weightIn) {
  return false
}
```

`WeightIn` and `targetWeight` is always equal, no check for the two targets having the same parents.

Remediation

We recommend replacing `targetWeight := target.Head.ParentWeight()` with `targetWeight := t.Head.ParentWeight()`. Furthermore, we recommend adding parents check for the two target tipsets.

Status

The Venus team has changed the assignment of `targetWeight` in accordance with the recommendation. The change is based on code which includes the check for equal parents.

Verification

Resolved.

Issue F: Program Panics if Length of Returned Messages is Less Than Length of Tipset Segments Requested

Location

`chainsync/syncer/syncer.go#L543-L548`

`chainsync/exchange/client.go#L165`

Synopsis

In syncing with target tipsets, the client must fetch messages (`fetchSegMessage`) for a segment of tipsets from a peer client. The returned message length could be less than the number of requested tipsets. When processing the returned message for each tipset in a loop over requested tipsets, the program will panic.

Impact

The program will panic and exit unexpectedly.

Preconditions

The length of the returned message is less than the number of requested tipsets in `GetChainMessages` when the syncing peer client is down or the network is down.

Feasibility

This is feasible and likely when preconditions are met (i.e. the syncing peer client is down or disconnected due to network problems).

Technical Details

In `fetchSegMessage`:

```
messages, err := syncer.exchangeClient.GetChainMessages(ctx, leftChain)

...

for index, tip := range leftChain {

  fts, err := zipTipSetAndMessages(bs, tip, messages[index].Bls,
messages[index].Secpk, messages[index].BlsIncludes,
messages[index].SecpkIncludes)

...

}
```

Message length might be less than the length of `leftChain`, in which case the loop over `leftChain` the program would panic upon reaching the out of range messages index.

Remediation

We recommend checking the length of the return message against the length of the requested tipsets (`leftChain`) before processing the messages of each tipset (`zipTipSetAndMessages`). If the length of the return message is less than the length of the requested tipset, process the partially returned messages from requested tipsets and put the unfetched tipset back in the tipset to be requested (`leftChain`) and refetch it from another peer.

Status

The Venus team has added a condition which returns an error if the length of the messages and tipset are not equal. We recommend implementing a change, which would add the unfetched tipset back to the queue to refetch from another peer, in order to fully resolve the issue.

Verification

Partially Resolved.

Issue G: Target Queue Size Exceeds the Preset Maximum Size

Location

`chainsync/types/target_tracker.go#L124`

Synopsis

When adding a target tipset to the target queue, an extra target will be added to the queue when queue size already reaches the preset max size (`bucket size`).

Impact
This would result in minimal impact to the functioning of the program, but it would cause confusion and inconsistency in the code.

Technical Details
```
 In add():

   if len(tq.q) <= tq.bucketSize {

      tq.q = append(tq.q, t)

    }
```

When `len(tq.q)== tq.bucketSize`, a new target will still be added to the queue`.`

Remediation
We recommend replacing .`len(tq.q) <= tq.bucketSize with len(tq.q) < tq.bucketSize`.

Status
The Venus team has resolved the off-by-one error.

Verification
Resolved.

Suggestions

Suggestion 1: Increase Code Comments and Disable Linting Exemption

Location
`build/main.go#L119`

Non-exhaustive examples:
`pkg/chain/store.go#L443`

`pkg/messagepool/gas.go#L217`

Synopsis
Functions that have a single call are documented, however, more complex functions are missing code comments. Without proper code comment coverage, the intended functionality of the code can be unclear to users and security researchers, which may result in misunderstanding the intended behavior of the system. In addition, the exemption of missing comments is enabled in the linter.

Mitigation
We recommend increasing code comment coverage with specific attention to complex functions and disabling the exemption of missing comments in the linter.

Status
The Venus team has added code comments to most exported types, functions, and variables.

However, instead of disabling the linting exemption, the Venus team has reconfigured the linter to make more exceptions instead of less. As a result, in addition to missing or malformed comments, it now ignores a collection of other issues. When disabling the exceptions in our test, our team found that all suppressed issues were related to missing or malformed comments, and not the newly added exceptions. As a result, we recommend that the Venus team fully disable the linting exemptions.

Verification

Partially Resolved.

Suggestion 2: Reduce Syncing Time

Location

`chainsync/syncer/syncer.go`

Synopsis

At present, a new node joining the network syncing with mainnet will take 2-3 seconds per tipset (22-30 days for a full sync and counting). Most of this time is spent on validating and executing messages in the block, which greatly reduces the syncing efficiency of the node in terms of time cost.

Mitigation

We recommend investigating and implementing an alternative syncing algorithm (e.g. parallel sync, snap sync) to reduce syncing time.

Status

The Venus team has responded that mitigating this suggestion requires protocol level changes to implement a fast syncing algorithm, which is complex to implement. At the time of this verification, there have not been any improvements in syncing time and the suggested mitigation remains..

Verification

Unresolved.

Suggestion 3: Increase Test Coverage

Location

`Filecoin-Venus`

Synopsis

A significant number of packages consisted of low or no test coverage. A robust test suite helps to identify errors and bugs that may lead to potential vulnerabilities.

Mitigation

We recommend increasing test coverage to include all packages, with tests for success and failure cases to account for edge cases, unexpected, and unintended behavior.

Status

The Venus team has added additional tests, but large parts of the code base remain under-tested. We recommend further increasing test coverage such that it includes all parts of the code base.

Verification

Partially Resolved.

Suggestion 4: Correct Inaccurate and Misleading Names

Location

`pkg/wallet/backend.go#L27-L29`

`pkg/wallet/dsbackend.go#L241`

`pkg/wallet/dsbackend.go#L272`

`pkg/wallet/wallet.go#L36`

`pkg/wallet/passphrase.go#L33`

`pkg/wallet/passphrase.go#L52`

`pkg/wallet/passphrase.go#L107`

`pkg/wallet/passphrase.go#L133`

`chainsync/syncer/syncer.go#L525`

`cmd/message.go#L123`

Synopsis

In multiple instances in the code, the variable and function naming convention results in increased complexity in understanding the intended functionality of the code.

For example:

- `Locked/UnLocked`: These are adjectives and intuitively imply that they return whether the wallet is in locked or unlocked state.
- `Kind`: A "kind" is a technical term in Go's reflect terminology, and in `wallet.go` it refers to a reflect type. This is not the same and is therefore misleading.
- `Auth`: In the encryption and decryption functions in `passphrase.go`, the password is referred to as "auth". "Auth" is commonly used as a function name but not as a variable name for "password".

Remediation

We recommend implementing and adhering to a single naming convention for variables and functions that is descriptive and accurate, and that is consistent with generally accepted best practices.

In reference to the examples above, we recommend:

- Using `LockWallet/UnlockWallet` instead of `Locked/UnLocked`. Don't use `Lock/Unlock` so it can be easily distinguished from the `sync.Locker` interface.
- Using the variable names `tipe` or `typ` to refer to types because "type" is a Go keyword, and can not be used.
- Using password or passphrase as a variable name when referring to passwords or passphrases.

Additionally, we found an instance of a wrong error message, "invalid gas limit". We recommend correcting this to "invalid nonce option".

Status

The Venus team has improved the naming conventions.

Verification

Resolved.

Suggestion 5: Remove Duplicate Code

Location

`pkg/crypto/bigint.go`

`pkg/types/bigint.go`

`pkg/crypto/bigint_test.go`

`pkg/types/bigint_test.go`

Synopsis

The repository contains duplicate code that was updated inconsistently. Duplicate code may lead to confusion and possible errors when updating and reviewing the code.

Remediation

We recommend removing instances of duplicate code by merging the features of the two versions and using only a single instance of the duplicate code.

Status

The Venus team has removed duplicate code from the `pkg/crypto` package and instead uses the code with equivalent functionality from `pkg/types`.

Verification

Resolved.

Suggestion 6: Use 32 Random Bytes as HMAC JSON Web Token Secret

Location

`pkg/jwtauth/jwt.go#L62-L68`

Synopsis

A Random RSA public key is used as an HMAC secret.

Technical Details

Message authentication codes (MACs) are a class of primitives that can be used to verify the authenticity of data in the presence of a shared secret. They could be seen as symmetric signatures. HMAC is a MAC algorithm often used by JSON Web Tokens (JWTs). RSA is an asymmetric cryptosystem that can be used to construct signatures and encryption. The code in question uses an RSA public key as an HMAC secret.

Remediation

We recommend using read 32 bytes from “crypto/rand” and use as key instead of generating a new RSA key pair.

Status

The Venus team updated the code to use 32 random bytes as the JWT key.

Verification

Resolved.

Suggestion 7: Check That Linting Works as Expected

Synopsis

While `golangci-lint` claims to run `staticcheck`, manually running `staticcheck` reports several potential issues that are not shown when running `build lint`.

Technical Details

Linting code consists of running a series of automated code quality checks. The implementation makes use of the `golangci-lint` tool, which is meant to run a collection of different linters on the code base. However, we found that running `staticcheck`, which appears to be run by `golangci-lint`, reports a number of issues that are not reported by `golangci-lint`, in addition to the issues disabled in the Venus implementation.

Remediation

We recommend ensuring that `golang-ci` properly runs `staticcheck`, or run `staticcheck` separately in the Continuous Integration (CI) pipeline. In addition, we recommend verifying that other linters are run as expected. To set up `staticcheck` linting in the `golangci` configuration, enable the linters `staticcheck`, `unused`, `gosimple` and `stylecheck`.

Status

The Venus team has enabled all necessary linters in `golangci-lint` and addressed the issues reported by the linters.

Verification

Resolved.

Suggestion 8: Switch Away From Deprecated Packages

Location

`app/submodule/network/libp2p.go#L15`

`pkg/testhelpers/test_daemon.go#L25`

`pkg/metrics/export.go#L10`

`app/node/node.go#L17`

`cmd/daemon_daemon_test.go/#L13`

`cmd/version_daemon_test.go#L13`

`cmd/main.go#L16`

Synopsis

The packages `go-multiaddr-net` and `go-libp2p-crypto` have both been deprecated by their authors and will not receive updates. Using deprecated dependencies that are no longer maintained may result in security vulnerabilities.

Remediation

We recommend replacing the existing deprecated packages with the following:

- `github.com/libp2p/go-libp2p-core/crypto` instead of `github.com/libp2p/go-libp2p-crypto`
- `github.com/multiformats/go-multiaddr/net` instead of `github.com/multiformats/go-multiaddr-net`

Status

The Venus team has updated the implementation so that it no longer depends on the deprecated packages.

Verification

Resolved.

Suggestion 9: Conduct an Audit of the Trusted Setup

Synopsis

The Venus implementation's Common Reference String (CRS) requires a trusted setup that utilizes a somewhat centralized MPC ceremony, with limited participants and transparency. While this system component is out of scope, it is worth noting that it introduces the possibility of collusion and compromising of the proof system. In addition, the system implements a Groth16 zk-SNARK proof, creating a one-time and non-updateable CRS.

Mitigation

We recommend that a security audit of the trusted setup be performed by an independent team, in order to explore ways to protect the network and users from compromise of the proof system.

Status

The Venus team has responded they will ask the Lotus team to provide trusted setup security documentation. These documents have not been provided at the time of the verification and, as a result, the suggested mitigation remains unresolved.

Verification

Unresolved.

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.