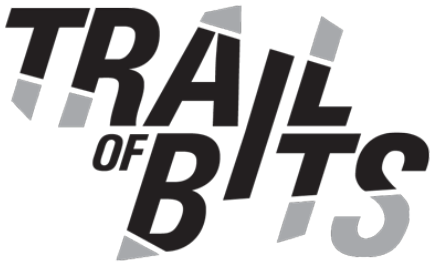

# Lighthouse

## Security Assessment

**December 18, 2020**


Prepared For:
Mehdi Zerouali | *Sigma Prime*
mehdi@sigmaprime.io

Prepared By:
Sam Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

Artur Cygan | *Trail of Bits*
artur.cygan@trailofbits.com

Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com


Changelog:

| | |
|---|---|
| June 1, 2020: | Initial report delivered |
| July 20, 2020: | Added Appendix C. Client's Responses to Findings |
| July 27, 2020: | Copyedited |
| August 24, 2020: | Updated Appendix C to Fix Log |
| October 9, 2020: | Updated to include review of network layer |
| October 19, 2020: | Copyedited |
| December 18, 2020: | Updated Fix Log to include network layer |

# Executive Summary

From May 18 through October 9, 2020, Sigma Prime engaged Trail of Bits to review the security of Lighthouse, an Ethereum 2.0 client. This assessment was conducted in two phases. In the first phase (May 18 through May 29, 2020), Trail of Bits reviewed the consensus layer, with four engineers devoting four person-weeks to commit `ddd63c0` of the Lighthouse repository. In the second phase, (October 5 through October 9, 2020) Trail of Bits reviewed the network layer, with three engineers devoting two person-weeks to commit `da44821` of the Lighthouse repository.

## Consensus Layer

Due to the length of the engagement and the complexity of the codebase, we focused on the areas that Sigma Prime considered to be the riskiest, specifically:

- Attestation processing
- The beacon chain caches
- Block processing
- Key management
- The operation pool
- Slashing protection
- SSZ decoding
- The BLS library

Within each of these areas, we paid particular attention to deadlock/concurrency issues, integer overflows, out-of-bounds array accesses, and error propagation/suppression issues.

During the first week, we verified that we could build the codebase, and that all unit tests passed. We ran `cargo-audit`, `cargo-upgrades`, and `cargo-clippy` over the codebase. We began manual review, focusing on the contents of the `beacon_node` directory (minus the `eth2-libp2p`, `network`, and `rest_api` subdirectories) and the `sigp/milagro_bls` library.

During the second week, we expanded our manual review beyond the `beacon_node` directory to include key management and non–BLS-related cryptographic code. We ran the unit tests with address sanitizer and thread sanitizer enabled. We also developed a custom analysis to look for unintentional state changes when errors occur.

Our efforts resulted in three high-, one medium-, three low-, and nine informational-severity findings. The three high-severity findings concern exposure of potentially sensitive information. Lesser findings concern cryptography, error reporting, a memory leak, and areas where the code could generally be updated.

Although using thread sanitizer did not produce any findings, it's worth mentioning. Using a command similar to that of Figure 8.2, we ran the unit tests with thread sanitizer enabled, and it produced multiple data race warnings. Most of these appear to be false positives due to the thread sanitizer's ignorance of certain synchronization primitives. There was one report concerning `rayon_core::registry::global_registry` that we could not rule out as a false positive. However, given the maturity of `rayon`'s codebase, and the nature of the bug considered, we directed our attention elsewhere.

Lighthouse bears the characteristics typical of a project under active development. At times, though, this made the code difficult to navigate. For example, the subject of this audit was the consensus layer, i.e., everything above the network layer. However, both consensus and networking code are intermingled within the `beacon_node` directory. Ideally, the project structure would reflect this conceptual division.

As a more specific example, we noticed that a mix of styles is used in object construction (TOB-LIHO-009). Ideally, one style would be used throughout. We recommend that such architectural issues be addressed early to avoid a major refactor later on.

As mentioned above, we focused on the areas that Sigma Prime considered to be the riskiest. However, due the length of the engagement and the complexity of the code, we feel that Lighthouse could benefit from additional security review. If budgetary constraints allow, we recommend that at least two additional engineer-weeks be spent on:

- The shuffling algorithm
- Fork choice
- Determination of the genesis block
- The validator client
- Extending existing fuzzing work
- Revisiting areas covered in this report, especially error-handling code

## Network Layer

In this phase of the engagement, we focused on the `eth2_libp2p` and `network` crates. We verified that the crates' unit tests passed, ran the unit tests under thread sanitizer, computed their code coverage, and checked them for unintentional state changes when errors occur (i.e., using the analysis that produced TOB-LIHO-006).

We also manually reviewed the code for compliance with the Ethereum 2.0 networking specification; verified correct use of the Noise XX protocol; and fuzzed all but two of the crates' `From` trait implementations and a few similar functions.

Our efforts resulted in one low-severity and three informational-severity findings. The one low-severity finding concerns a bug in a `rust-libp2p` deserialization function. The three

informational findings concern unit test code coverage and two minor deviations from the Ethereum 2.0 specification.

In addition to the four new findings, we wrote an appendix detailing how we fuzzed the `eth2_libp2p` and `network` crates' `From` trait implementations.

Our main recommendation is to write additional unit tests for the `eth2_libp2p` and `network` crates. Unit tests help expose errors, provide a sort of documentation of the code, and exercise the code in a more systematic way than any human can, which helps protect against regressions. Finally, we believe additional unit tests could have revealed more results through our analyses, e.g., fuzzing and looking for unintentional state changes when errors occur.

# Project Dashboard

**Application Summary**

| | |
|---|---|
| Name | Lighthouse |
| Version | ddd63c0de146133ce7877bee218710a2b41defd8<br>da44821e39018a7b480f6ab3ef398776e63446bb |
| Type | Rust |
| Platforms | POSIX |

**Engagement Summary**

| | |
|---|---|
| Dates | May 18–May 29, 2020<br>October 5–October 9, 2020 |
| Method | Whitebox |
| Consultants Engaged | 4 |
| Level of Effort | 6 person-weeks |

**Vulnerability Summary**

| | | |
|---|---|---|
| Total High-Severity Issues | 3 | ■ ■ ■ |
| Total Medium-Severity Issues | 1 | ■ |
| Total Low-Severity Issues | 4 | ■ ■ ■ ■ |
| Total Informational-Severity Issues | 12 | ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ |
| Total | 20 | |

**Category Breakdown**

| | | |
|---|---|---|
| Configuration | 1 | ■ |
| Cryptography | 3 | ■ ■ ■ |
| Data Exposure | 3 | ■ ■ ■ |
| Data Validation | 1 | ■ |
| Denial of Service | 4 | ■ ■ ■ ■ |
| Error Reporting | 3 | ■ ■ ■ |
| Patching | 4 | ■ ■ ■ ■ |
| Undefined Behavior | 1 | ■ |
| Total | 20 | |

## Engagement Goals

The engagement was scoped to provide a security assessment of the consensus and network layers of the Lighthouse repository.

Specifically, we sought to answer the following questions:

- Are deadlocks or other concurrency issues possible?
- Are integer overflows possible?
- Is out-of-bounds array access possible?
- Are there error propagation issues, e.g., are errors unintentionally suppressed?
- Can remote code execution be achieved through the network layer?
- Is it possible to cause a Lighthouse node to miss packets?
- Are any denial-of-service vulnerabilities present in the code base?

## Coverage

The project as a whole was analyzed with `cargo-audit` and `cargo-upgrades`. In addition, the following specific components were examined.

**Attestation processing.** Manually reviewed. Subject to static analysis by `cargo-clippy`. Unit tests verified to pass. Unit tests run with address sanitizer and thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage.

**Beacon chain caches.** Manually reviewed. Subject to static analysis by `cargo-clippy`. Unit tests verified to pass. Unit tests run with address sanitizer and thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage.

**Block processing.** Manually reviewed. Subject to static analysis by `cargo-clippy`. Unit tests verified to pass. Unit tests run with address sanitizer and thread sanitizer enabled. Unit tests checked for state changes upon error (see TOB-LIHO-006). Unit tests reviewed for code coverage.

**Key management.** Manually reviewed.

**Operation Pool.** Manually reviewed. Subject to static analysis by `cargo-clippy`. Unit tests verified to pass. Unit tests run with address sanitizer and thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage.

**Slashing protection.** Manually reviewed. Subject to static analysis by `cargo-clippy`. Unit tests verified to pass. Unit tests run with address sanitizer and thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage.

**SSZ decoding.** Manually reviewed. Subject to static analysis by `cargo-clippy`. Unit tests verified to pass. Unit tests run with address sanitizer and thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage.

**sigp/milagro_bls.** A brief review of `sigp/milagro_bls`, its upstream dependency `incubator`, and their usage in Lighthouse was conducted for implementation correctness and common pitfalls. Apart from a lack of tests in `milagro_bls`, of which there are plenty in the `incubator` module, there was nothing of note to report. In the time spent reviewing the implementation of the curve and pairing function, we found no issues.

**`eth2_libp2p` crate.** Manually reviewed. Unit tests verified to pass. Unit tests run with thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage. All but two `From` trait implementations and some similar functions fuzzed. Verified correct use of the Noise XX protocol.

**`network` crate.** Manually reviewed. Unit tests verified to pass. Unit tests run with thread sanitizer enabled. Unit tests checked for state changes upon error. Unit tests reviewed for code coverage. `From` trait implementation and some similar functions fuzzed.

# Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

## Short Term

❏ **Upgrade `rusqlite` from `0.22.0` to `0.23.1`. Patch crate `r2d2_sqlite` to use `rusqlite 0.23.1`, and use the patched version.** These steps will eliminate potential memory corruption vulnerabilities. TOB-LIHO-001

❏ **Update dependencies to the latest version wherever possible.** Using out-of-date dependencies could mean critical bug fixes are missed. TOB-LIHO-002

❏ **Adjust the `Decode` macro so that decoding completes prior to struct initialization.** Doing so will eliminate potential undefined behavior. TOB-LIHO-003

❏ **Remove all legacy code from the codebase before considering the code production-ready, and ensure that documentation is up to date and accurate.** These steps will reduce the risk of using legacy code improperly, and will facilitate onboarding of new developers. TOB-LIHO-004

❏ **Add a `sha256` checksum to the build code and validate the downloaded file with it.** Doing so will reduce the risk of an attacker gaining control of this file. TOB-LIHO-005

❏ **Adjust the implementation of `process_block_header`, `process_proposer_slashings`, and `process_exits` so they do not modify the state argument upon error.** This will reduce the likelihood of an error having an unintentional, lasting effect on the program's state. TOB-LIHO-006

❏ **Add an error type to represent multiple errors, and adjust the implementation of leaves so that it returns an instance of this "multiple error" error type.** Eliminating the use of `ParallelIterator::collect` will help ensure that errors are not suppressed and that valuable system information is not lost. TOB-LIHO-007

❏ **Re-implement the shutdown as described above to eliminate memory leaks.** TOB-LIHO-008

❏ **Adopt one pattern, be it the "Builder Pattern," the "Init Struct Pattern," or something else, and try to adhere to it.** This will make your code easier to read, understand, and reason about. TOB-LIHO-009

❏ **Ensure that nobody ever touches the IV length.** Setting the IV length to anything other than 16 could introduce undefined behavior. TOB-LIHO-010

❏ **Consider revising EIP-2335 to include `argon2id` as a possible KDF.** As argued, `argon2id` is a better choice than scrypt or PBKDF2. Incorporating `argon2id` into EIP-2335 will provide better security to the community as a whole. TOB-LIHO-011

❏ **Consider revising EIP-2333 so that the `mod_r` operation becomes uniformly distributed, or can fail.** Adjusting the EIP will provide better security to the community as a whole. TOB-LIHO-012

❏ **Change the code to eliminate panics in the indicated places.** This will eliminate a potential denial-of-service vector. TOB-LIHO-013

❏ **Encrypt the validator key with a password and require the user to enter the password on Lighthouse startup.** This will eliminate the need to store sensitive information in plaintext on the filesystem. TOB-LIHO-014

❏ **Require the user to enter the wallet password on Lighthouse startup.** This will eliminate the need to store sensitive information in plaintext on the filesystem. TOB-LIHO-015

❏ **Remove the option to pass `p2p-priv-key` from the command line.** Passing this option on the command line makes it accessible to an attacker. TOB-LIHO-016

❏ **Add unit tests for `eth2_libp2p` and `network` crate functions not currently exercised by unit tests.** Ideally, there will be at least one test for each "happy" (successful) path, and at least one test for each "sad" (failing) path. A comprehensive set of unit tests will help expose errors, protect against regressions, and provide a sort of documentation to users. TOB-LIHO-017

❏ **Patch the code in Figure 18.1 by limiting the size of the value passed to `Vec::with_capacity`.** Use the patched version of `rust-libp2p` until the bug is fixed upstream. These steps will eliminate a potential denial-of-service attack. TOB-LIHO-018

❏ **When reading SSZ encoded data, limit the size of the read buffer to `max_compressed_len`.** This will make a subsequent check against `max_compressed_len` unnecessary and will bring Lighthouse more in line with the Ethereum 2.0 specification. TOB-LIHO-019

❏ **Adjust either the implementation or the specification so that the parameter choices match and desired network performance is achieved.** TOB-LIHO-020

## Long Term

❏ **Regularly run `cargo-audit` over the codebase to help reveal similar bugs.** TOB-LIHO-001

❏ **Regularly run `cargo-upgrades` and `cargo update --dry-run` over the codebase to ensure that the project stays up to date with its dependencies.** TOB-LIHO-002

❏ **Regularly run `cargo-clippy` over the codebase to help reveal similar bugs**. TOB-LIHO-003

❏ **Ensure that deprecated code is tracked in an issue-tracking system so it's not used in production.** TOB-LIHO-004

❏ **Validate all important files downloaded from the Internet with a checksum.** This will help ensure that the application functions correctly following an unexpected content change, be it malicious or otherwise. TOB-LIHO-005

❏ **Use mutability only where absolutely necessary, and regularly scrutinize those uses to help ensure that bugs are not introduced as a result of mutability.** TOB-LIHO-006

❏ **Produce verbose logs in tests, and review those logs regularly for changes to identify unintended sources of non-determinism.** TOB-LIHO-007

❏ **Regularly run your tests with address sanitizer enabled.** If a program is currently free of memory leaks, doing this can help ensure memory leaks are not introduced. TOB-LIHO-008

❏ **As new components are added to the system, try to use the same pattern in their construction.** This will reduce cognitive load both on developers adding components to the system and on users trying to understand the system. TOB-LIHO-009

❏ **Consider switching to a more actively maintained cryptography library with better guarantees, like `RustCrypto/block-ciphers`.** TOB-LIHO-010

❏ **If EIP-2335 is revised, implement the use of `argon2id` as a KDF within `eth2_keystore`.** This will provide better security to the derived keys. TOB-LIHO-011

❏ **Implement any changes accepted to EIP-2333 to reduce the risk of a secret key compromise.** TOB-LIHO-012

❏ **Think about further minimizing panic occurrences throughout the codebase.** Also, consider using `no-panic` to reduce the likelihood of a panic becoming reachable in future revisions. TOB-LIHO-013

❏ **Give users the option of storing their credentials in an HSM (Hardware Security Module) or password manager such as Vault.** This will allow the wallet to offload responsibility for sensitive material to a service designed to handle such information. TOB-LIHO-014 and TOB-LIHO-015

❏ **As new command line options are added to the system, ensure that they do not involve sensitive information.** Command line options are generally accessible to an attacker and could reveal such information. TOB-LIHO-016

❏ **Ensure unit tests are written for new functions as they are added to the codebase to prevent them from introducing bugs.** TOB-LIHO-017

❏ **Consider adding the fuzz targets in Appendix D to the set of targets that you regularly fuzz.** This bug was found by fuzzing one of those targets, so regularly fuzzing them could help identify similar bugs, especially as Lighthouse's dependencies evolve. TOB-LIHO-018

❏ **Adhere to the specification unless there is a good reason not to do so.** This will help Lighthouse avoid problems already anticipated by the specification's authors. TOB-LIHO-019

❏ **If changes are made to any of these parameters, change both the implementation and specification simultaneously** so they maintain parity as they each evolve. TOB-LIHO-020

## Findings Summary

| # | Title | Type | Severity |
|---|---|---|---|
| 1 | Codebase uses a crate with a RUSTSEC advisory | Patching | Medium |
| 2 | Build process relies on outdated dependencies | Patching | Informational |
| 3 | Assumptions about struct field initialization order | Undefined Behavior | Low |
| 4 | Comments suggest code and documentation are out of date | Patching | Informational |
| 5 | Downloaded deposit contract is not validated with a checksum | Data Validation | Informational |
| 6 | `BeaconState` objects are mutated upon error | Error Reporting | Informational |
| 7 | Errors produced by "`ParallelValidatorTreeHash::leaves`" are non-deterministic | Error Reporting | Informational |
| 8 | Memory leak due to non-graceful shutdown | Denial of Service | Low |
| 9 | Builder pattern is not strictly adhered to | Patching | Informational |
| 10 | `rust-crypto` is unmaintained and a better alternative should be used | Cryptography | Informational |
| 11 | Consider using `argon2id` as a KDF | Cryptography | Informational |
| 12 | Bias in BLS secret key generation | Cryptography | Low |
| 13 | Unnecessary use of panicking functions | Denial of Service | Informational |

| | | | |
|---|---|---|---|
| 14 | Password to validator private key is stored in plaintext | Data Exposure | High |
| 15 | Password to wallet is stored in plaintext | Data Exposure | High |
| 16 | Secret key passed as CLI argument | Data Exposure | High |
| 17 | Insufficient network layer unit tests | Error Reporting | Informational |
| 18 | Memory exhaustion via `Multiaddr` deserialization | Denial of Service | Low |
| 19 | SSZ snappy decoder reads more data than specification recommends | Denial of Service | Informational |
| 20 | `Gossipsub` parameters deviate from the specification | Configuration | Informational |

## 1. Codebase uses a crate with a RUSTSEC advisory

Severity: Medium
Type: Patching
Target: `rusqlite 0.22.0`

Difficulty: Undetermined
Finding ID: TOB-LIHO-001

**Description**
The Lighthouse repository makes use of a crate with a Rust Security (RUSTSEC) advisory. Specifically, the crate `rusqlite 0.22.0` contains vulnerabilities that could lead to memory corruption.

The `rusqlite 0.23.0` release states:

> The release primarily contains a number of security/memory safety fixes...[that] mostly impact APIs exposed through `features`, so while there are a lot of them, if you're using `rusqlite` under default features, you're fine.

Note that the `consensus/types` and `validator_client/slashing_protection` crates use the "`bundled`" feature, and the `r2d2_sqlite` crate uses the "`trace`" feature.

In addition to the vulnerabilities mentioned above, `cargo-audit` produces five warnings. Several of them concern crates that are no longer maintained.

**Exploit Scenario**
Eve discovers a code path leading to the vulnerable crate. Eve uses this code path to crash nodes, corrupt memory, etc.

**Recommendations**
Short term, upgrade `rusqlite` from `0.22.0` to `0.23.1`. Patch crate `r2d2_sqlite` to use `rusqlite 0.23.1`, and use the patched version. These steps will eliminate potential memory corruption vulnerabilities.

Long term, regularly run `cargo-audit` over the codebase to help reveal similar bugs.

**References**
- RUSTSEC-2020-0014: `rusqlite`: Various memory safety issues
- Release `rusqlite 0.23.0`, `libsqlite-sys 0.18.0`
- `cargo-audit`

## 2. Build process relies on outdated dependencies

Severity: Informational
Type: Patching
Target: `Cargo.toml`, `Cargo.lock`

Difficulty: Undetermined
Finding ID: TOB-LIHO-002

**Description**
Updated versions of many of Lighouse's dependencies are available. Since silent bug fixes are common, the dependencies should be reviewed and updated wherever possible.

| Dependency | Version currently in use | Latest version available |
|---|---|---|
| `db-key` | `0.0.5` | `0.1.0` |
| `enr` | `0.1.0-alpha.7` | `0.1.0` |
| `hex` | `0.3.2` | `0.4.2` |
| `libp2p` | `0.18.1` | `0.19.0` |
| `libp2p-tcp` | `0.18.0` | `0.19.0` |
| `lru` | `0.4.3` | `0.4.4` |
| `miow` | `0.3.3` | `0.3.4` |
| `parking_lot` | `0.9.0` | `0.10.2` |
| `prometheus` | `0.8.0` | `0.9.0` |
| `rusqlite` (TOB-LIHO-001) | `0.22.0` | `0.23.1` |
| `syn` | `1.0.22` | `1.0.23` |
| `web3` (see below) | `b6c81f97` | `a3e5a53` |

Note that the project does not build with the latest commit of the `web3` dependency. However, the commit currently used is not specified in the `Cargo.toml` file. This makes the `Cargo.lock` file necessary to build the project. Reliance on the `Cargo.lock` file should be avoided, if possible.

To be clear, we are not suggesting that you abandon use of the `Cargo.lock` file, only that it should not be relied upon.

**Exploit Scenario**

Eve learns of an exploitable vulnerability in an old version of a dependency. Eve forks the Lighthouse repository and modifies the code in a way that looks benign, but actually exploits the machines of the developers who download and build the fork.

**Recommendations**
Short term, update dependencies to the latest version wherever possible. Using out-of-date dependencies could mean critical bug fixes are missed.

Long term, regularly run `cargo-upgrades` and `cargo update --dry-run` over the codebase to ensure that the project stays up to date with its dependencies.

**References**
- `cargo-upgrades`
- `cargo update`

## 3. Assumptions about struct field initialization order

Severity: Low
Type: Undefined Behavior
Target: `consensus/ssz_derive/src/lib.rs`

Difficulty: High
Finding ID: TOB-LIHO-003

**Description**
The implementation of the `Decode` macro makes assumptions about the order in which struct fields are initialized. Such assumptions are not part of the Rust specification and could break on certain platforms or in newer versions of the Rust compiler.

The relevant portions of the `Decode` macro appear in Figures 3.1 and 3.2. Note the use of the local variables `start` and `end` in Figure 3.2.

```
fixed_decodes.push(quote! {
    #ident: decode_field!(#ty)
});
```

*Figure 3.1:* consensus/ssz_derive/src/lib.rs#L209-L211.

```
macro_rules! decode_field {
    ($type: ty) => {{
        start = end;
        end += <$type as ssz::Decode>::ssz_fixed_len();
        let slice = bytes.get(start..end)
            .ok_or_else(|| ssz::DecodeError::InvalidByteLength {
                len: bytes.len(),
                expected: end
            })?;
        <$type as ssz::Decode>::from_ssz_bytes(slice)?
    }};
}

Ok(Self {
    #(
        #fixed_decodes,
    )*
})
```

*Figure 3.2:* consensus/ssz_derive/src/lib.rs#L259-L276.

Sample output of the `Decode` macro appears in Figure 3.3. Again, note the use of the local variables `start` and `end`.

```
Ok(Self {
    backing: {
```

```
                    start = end;
                    end += <Vec<T> as ssz::Decode>::ssz_fixed_len();
                    let slice = bytes.get(start..end).ok_or_else(|| {
                        ssz::DecodeError::InvalidByteLength {
                            len: bytes.len(),
                            expected: end,
                        }
                    })?;
                    <Vec<T> as ssz::Decode>::from_ssz_bytes(slice)?
                },
                offsets: {
                    start = end;
                    end += <Vec<usize> as ssz::Decode>::ssz_fixed_len();
                    let slice = bytes.get(start..end).ok_or_else(|| {
                        ssz::DecodeError::InvalidByteLength {
                            len: bytes.len(),
                            expected: end,
                        }
                    })?;
                    <Vec<usize> as ssz::Decode>::from_ssz_bytes(slice)?
                },
            })
```

*Figure 3.3: The result of applying the* `Decode` *macro to* struct CacheArena. *(Produced using* `cargo-expand`*.)*

A closely related problem occurs with respect to error production. Consider the code in Figure 3.4. The initializers for both `proposer_index` and `attestations` could generate errors. An error that is produced depends upon the order in which those fields are initialized. (See also TOB-LIHO-007.)

```
        let mut block = SignedBeaconBlock {
            message: BeaconBlock {
                slot: state.slot,
                proposer_index: state.get_beacon_proposer_index(state.slot, &self.spec)? as
u64,
                parent_root,
                state_root: Hash256::zero(),
                body: BeaconBlockBody {
                    randao_reveal,
                    eth1_data,
                    graffiti,
                    proposer_slashings: proposer_slashings.into(),
                    attester_slashings: attester_slashings.into(),
                    attestations: self
                        .op_pool
                        .get_attestations(&state, attestation_filter, &self.spec)
                        .map_err(BlockProductionError::OpPoolError)?
                        .into(),
```

```
                    deposits,
                    voluntary_exits: self.op_pool.get_voluntary_exits(&state,
&self.spec).into(),
                },
            },
            // The block is not signed here, that is the task of a validator client.
            signature: Signature::empty_signature(),
        };
```

*Figure 3.4:* beacon_node/beacon_chain/src/beacon_chain.rs#L1639-L1662.

**Exploit Scenario**

A future version of the Rust compiler initializes fields in an order that is not their lexical order. Lighthouse clients built with this version of the Rust compiler are unable to interoperate with other nodes.

**Recommendations**

Short term, adjust the `Decode` macro so that decoding completes prior to struct initialization. Doing so will eliminate potential undefined behavior.

Long term, regularly run `cargo-clippy` over the codebase to help reveal similar bugs.

**References**

- cargo-expand

## 4. Comments suggest code and documentation are out of date

Severity: Informational
Type: Patching
Target: Various

Difficulty: Not applicable
Finding ID: TOB-LIHO-004

**Description**
In several places, comments suggest that the code and/or documentation need to be updated. Legacy code is not just a maintenance burden, it also risks being used when it should not be. Out-of-date documentation makes navigating the code more difficult and discourages new developers.

Examples of comments suggesting deprecation appear in Figures 4.1–4.3.

```
        // TODO: currently we do not check the FFG source/target. This is what the spec
dictates
        // but it seems wrong.
        //
        // I have opened an issue on the specs repo for this:
        //
        // https://github.com/ethereum/eth2.0-specs/issues/1636
        //
        // We should revisit this code once that issue has been resolved.
```

*Figure 4.1:* beacon_node/beacon_chain/src/attestation_verification.rs#L554-L561.
*Issue 1636 on* `eth2.0-specs` *is closed.*

```
/// This is a legacy object that is being kept around to reduce merge conflicts.
///
/// TODO: As soon as this is merged into master, it should be removed as soon as possible.
#[derive(Debug, PartialEq)]
pub enum BlockProcessingOutcome {
```

*Figure 4.2:* beacon_node/beacon_chain/src/block_verification/block_processing_outcome.rs#L5-L9.
*The code has been merged into master.*

```
# Lighthouse Validator Client

The Validator Client (VC) is a stand-alone binary which connects to a Beacon
Node (BN) and fulfils the roles of a validator.
```

*Figure 4.3:* validator_client/README.md#L1-L4.
*The validator client resides within the same binary as, e.g., the* `account_manager`*, whose README features a similar message.*

**Exploit Scenario**
A block validation function is updated to protect against a new bug class discovered post-deployment. However, code using the legacy version of that function remains vulnerable, and is exploited.

**Recommendations**
Short term, remove all legacy code from the codebase before considering the code production-ready, and ensure that documentation is up to date and accurate. These steps will reduce the risk of using legacy code improperly, and will facilitate onboarding of new developers.

Long term, ensure that deprecated code is tracked in an issue-tracking system so it's not used in production.

## 5. Downloaded deposit contract is not validated with a checksum

Severity: Informational
Type: Data Validation
Target: `common/deposit_contract/build.rs`

Difficulty: High
Finding ID: TOB-LIHO-005

**Description**
ABI and bytecode for the deposit contract are downloaded from the `ethereum/eth2.0-specs` GitHub repository. While the request is called over https, the file is not verified with a checksum. There is a small risk of the file being overwritten, allowing an adversary to swap the deposit contract to a rogue one.

**Exploit Scenario**
Eve gains push permission to the `ethereum/eth2.0-specs` GitHub repository. She swaps the deposit contract to a rogue one, which might lead to node malfunctioning and could risk a slashing penalty.

**Recommendations**
Short term, add a `sha256` checksum to the build code and validate the downloaded file with it. Doing so will reduce the risk of an attacker gaining control of this file.

Long term, validate all important files downloaded from the Internet with a checksum. This will help ensure that the application functions correctly following an unexpected content change, be it malicious or otherwise.

## 6. `BeaconState` objects are mutated upon error

Severity: Informational | Difficulty: Undetermined
Type: Error Reporting | Finding ID: TOB-LIHO-006
Target: `consensus/state_processing/src/per_block_processing.rs`

**Description**
The functions `process_block_header`, `process_proposer_slashings`, and `process_exits` each take a reference to a mutable `BeaconState` object as their first argument. The argument is mutated even when an error occurs in those functions. Such a practice is error-prone and could have unintentional, lasting effects on the program's state.

The relevant code from `process_block_header` appears in Figures 6.1 and 6.2. The function sets the state's `latest_block_header` field (Figure 6.1). That change persists even if, for instance, an error occurs within the `verify!` macro (Figure 6.2).

```
    state.latest_block_header = block.temporary_block_header();

    // Verify proposer is not slashed
    let proposer = &state.validators[proposer_index];
    verify!(
        !proposer.slashed,
        HeaderInvalid::ProposerSlashed(proposer_index)
    );
```

*Figure 6.1:* consensus/state_processing/src/per_block_processing.rs#L180-L187.

```
macro_rules! verify {
    ($condition: expr, $result: expr) => {
        if !$condition {
            return
Err(crate::per_block_processing::errors::BlockOperationError::invalid($result));
        }
    };
}
```

*Figure 6.2:* consensus/state_processing/src/macros.rs#L1-L7.

A similar phenomenon exists with regard to `SszDecoder::decode_next`. Specifically, the function removes the next item even when it cannot be decoded. However, such behavior may be intentional.

```
    pub fn decode_next<T: Decode>(&mut self) -> Result<T, DecodeError> {
        T::from_ssz_bytes(self.items.remove(0))
    }
```

*Figure 6.3:* consensus/ssz/src/decode.rs#L274-L276.

**Exploit Scenario**
An error occurs during normal operation of Alice's Lighthouse node. The error has an unintentional, lasting effect on the program's state, and Alice is no longer able to sync with the network.

**Recommendations**
Short term, adjust the implementation of `process_block_header`, `process_proposer_slashings`, and `process_exits` so they do not modify the `state` argument upon error. This will reduce the likelihood of an error having an unintentional, lasting effect on the program's state.

Long term, use mutability only where absolutely necessary, and regularly scrutinize those uses to help ensure that bugs are not introduced as a result of mutability.

## 7. Errors produced by "`ParallelValidatorTreeHash::leaves`" are non-deterministic

Severity: Informational
Type: Error Reporting
Target: `consensus/types/src/beacon_state/tree_hash_cache.rs`

Difficulty: High
Finding ID: TOB-LIHO-007

**Description**
The function `ParallelValidatorTreeHash::leaves` can produce different errors when given the same inputs. Thus, some errors could be suppressed by less severe ones.

Specifically, `leaves` uses `ParallelIterator::collect` to prepare its return value. A section of `collect`'s documentation appears in Figure 7.1.

```
/// If any item is `Err`, then all previous `Ok` items collected are
/// discarded, and it returns that error.  If there are multiple errors, the
/// one returned is not deterministic.
```

*Figure 7.1:* src/result.rs#L90-L92.

Multiple types of errors can be generated in `leaves`, e.g., `TreeHashCacheInconsistent` and `CachedTreeHashError(...)`. If more than one error is generated, there is no guarantee as to which will be returned.

```
        self.arenas
            .par_iter_mut()
            .enumerate()
            .map(|(arena_index, (arena, caches))| {
                caches
                    .iter_mut()
                    .enumerate()
                    .map(move |(cache_index, cache)| {
                        let val_index = (arena_index * VALIDATORS_PER_ARENA) + cache_index;

                        let validator = validators
                            .get(val_index)
                            .ok_or_else(|| Error::TreeHashCacheInconsistent)?;

                        validator
                            .recalculate_tree_hash_root(arena, cache)
                            .map_err(Error::CachedTreeHashError)
                    })
                    .collect()
            })
            .collect()
```

*Figure 7.2:* consensus/types/src/beacon_state/tree_hash_cache.rs#L295-L315.

Note: We have not observed non-deterministic error production experimentally. However, the possibility of non-deterministic behavior is likely worth addressing.

**Exploit Scenario**
Alice runs a Lighthouse node and dismisses its error reports because they seem insignificant. In reality, the reports conceal more severe errors.

**Recommendations**
Short term, add an error type to represent multiple errors, and adjust the implementation of `leaves` so that it returns an instance of this "multiple error" error type. Eliminating the use of `ParallelIterator::collect` will help ensure that errors are not suppressed and that valuable system information is not lost.

Long term, produce verbose logs in tests, and review those logs regularly for changes to identify unintended sources of non-determinism.

## 8. Memory leak due to non-graceful shutdown

Severity: Low
Type: Denial of Service
Target: Various

Difficulty: Undetermined
Finding ID: TOB-LIHO-008

**Description**
Lighthouse nodes fail to clean up memory prior to shutdown. Such behavior could conceal more severe memory leaks and lead to resource exhaustion.

A comment in the definition of the `Client` struct suggests that all services are expected to shut down gracefully (see Figure 8.1).

```
   /// Exit channels will complete/error when dropped, causing each service to exit
gracefully.
   _exit_channels: Vec<tokio::sync::oneshot::Sender<()>>,
```

*Figure 8.1:* beacon_node/client/src/lib.rs#L28-L29.

However, the claim does not appear to hold. Specifically, running the `http_server_genesis_state` test with address sanitizer enabled (Figure 8.2) produces multiple memory leak errors.

```
RUSTFLAGS='-Z sanitizer=address' cargo +nightly test --target
x86_64-unknown-linux-gnu http_server_genesis_state
```

*Figure 8.2: Command to run the* `http_server_genesis_state` *test with address sanitizer enabled.*

While the specifics vary, the majority of the errors mention the `tokio` asynchronous IO framework.

In our experiments, the memory leak errors went away when we re-implemented the shutdown by:

- Having the `tokio` tasks listen on their respective exit channels, and exit upon receipt of a message.
- Storing the `JoinHandles` produced by `tokio::spawn` in the `Client`.
- Implementing the `Drop` trait for the `Client` so it sends on each exit channel and waits on each `JoinHandle`.

**Exploit Scenario**
Alice is a Lighthouse developer, and runs the tests with address sanitizer enabled. Alice dismisses the memory leak reports, believing they concern only node shutdown. However,

more severe reports that she does not notice “hide in the noise.” The memory leak leads to wide scale disruption of Lighthouse nodes.

**Recommendations**
Short term, re-implement the shutdown as described above to eliminate memory leaks.

Long term, regularly run your tests with address sanitizer enabled. If a program is currently free of memory leaks, doing this can help ensure memory leaks are not introduced.

**References**
- The Rust Programming Language: Graceful Shutdown and Cleanup (which does not use `tokio`, but is a useful reference, nonetheless)

## 9. Builder pattern is not strictly adhered to

Severity: Informational
Type: Patching
Target: Various

Difficulty: Not applicable
Finding ID: TOB-LIHO-009

**Description**
Lighthouse node components are constructed using a mix of different software patterns. The mixed style creates unnecessary complexity and makes the code more difficult to understand.

Consider the construction of `ProductionBeaconNode`, which inherits from `ProductionClient`. A `ProductionBeaconNode` is constructed by calling its `new` function, which takes a `client_config` parameter. Members of the `client_config` are passed to a builder to construct the `ProductionClient` (see Figure 9.1).

```
    pub async fn new(
        context: RuntimeContext<E>,
        mut client_config: ClientConfig,
    ) -> Result<Self, String> {
        let http_eth2_config = context.eth2_config().clone();
        let spec = context.eth2_config().spec.clone();
        let client_config_1 = client_config.clone();
        let client_genesis = client_config.genesis.clone();
        let store_config = client_config.store.clone();
        let log = context.log.clone();

        let db_path = client_config.create_db_path()?;
        let freezer_db_path_res = client_config.create_freezer_db_path();

        let builder = ClientBuilder::new(context.eth_spec_instance.clone())
            .runtime_context(context)
            .chain_spec(spec)
            .disk_store(&db_path, &freezer_db_path_res?, store_config)?
            .background_migrator()?;
```

*Figure 9.1:* beacon_node/src/lib.rs#L69-L87.

GeeksforGeeks lists the following as an advantage of the builder pattern:

> The parameters to the constructor are reduced and are provided in highly readable method calls.

Using a `client_config` parameter nullifies this advantage, since each element of the `client_config` must be used in order to have an effect. Using the `client_config` better resembles what Jacob Kiesel calls the "Init Struct Pattern."

A secondary problem concerns side effects. For a `ProductionClient`, many such side effects occur prior to the call of its builder's `build` function (see Figure 9.2). Such an approach deviates from the Rust style guide on how to apply the builder pattern.

```
    /// Immediately starts the networking stack.
    pub fn network(mut self, config: &NetworkConfig) -> Result<Self, String> {
```

*Figure 9.2:* beacon_node/client/src/builder.rs#L214-L215.

**Exploit Scenario**
Alice, a new Lighthouse developer, adds a component to the system. Confused by the mix of styles used to construct objects, Alice introduces a bug into the system.

**Recommendations**
Short term, adopt one pattern, be it the "Builder Pattern," the "Init Struct Pattern," or something else, and try to adhere to it. This will make your code easier to read, understand, and reason about.

Long term, as new components are added to the system, try to use the same pattern in their construction. This will reduce cognitive load both on developers adding components to the system and on users trying to understand the system.

**References**
- GeeksforGeeks: Builder Design Pattern
- X Blog: Init Struct Pattern
- Learn Rust: The builder pattern

## 10. `rust-crypto` is unmaintained and a better alternative should be used

Severity: Informational
Type: Cryptography
Target: `eth2_keystore`, `eth2_wallet`

Difficulty: Not applicable
Finding ID: TOB-LIHO-010

**Description**
The `eth2_keystore` module and consequently the `eth2_wallet` module depend on `rust-crypto` for their `aes-128-ctr` keystream. In using this keystream, we found a variety of bugs that result in footguns which Lighthouse currently avoids in these modules. Specifically, an unchecked counter leads to undefined behavior.

On `x86` or `x86_64` targets, `rust-crypto` attempts to use its optimized AESNI implementation instead of the general software implementation. This just converts the input IV array to a vector and constructs the implementation. The implementation itself is quite dumb and maintains a copy of this vector as the counter. This counter is incremented via the `add_ctr` function, which starts at the last byte of the vector, adds 1 to it, and performs carry operations as necessary until it passes the first byte of the vector. Since the beginning of the counter vector is passed to the encrypt function, counters do not increment correctly when their vector length is greater than 16, and counters will cycle too early if their vector length is less than 16, e.g., `[255] -> [0]` instead of `[0, ..., 255] -> [0, ..., 1, 0]`.

Because the start of the counter is passed to the encrypt function, which expects a full block of 16 bytes, there is the issue of uninitialized memory. The underlying encrypt function just performs the encryption, no questions asked, and the buffers are passed verbatim, leading to undefined behavior from the uninitialized memory when the IV length is less than 16.

The Lighthouse code currently sets the length to 16 in the keystore and wallet, which avoids all these dangerous code paths, but we are including this notice to warn the developers.

**Exploit Scenario**
An unwary developer changes the IV length in a fork or a future commit, thereby reducing the security of the encrypted keystore and leaking information about their secret key or rendering their secret key unrecoverable due to undefined behavior.

**Recommendations**
Short term, ensure that nobody ever touches the IV length. Setting the IV length to anything other than 16 could introduce undefined behavior.

Long term, consider switching to a more actively maintained cryptography library with better guarantees, like `RustCrypto/block-ciphers`.

## 11. Consider using `argon2id` as a KDF

Severity: Informational
Type: Cryptography
Target: `eth2_keystore`, `eth2_wallet`, `EIP-2335`

Difficulty: Not applicable
Finding ID: TOB-LIHO-011

**Description**
EIP-2335 is currently in the draft phase and supports two KDFs: PBKDF2 and scrypt. PBKDF2 has been around for a long time, and its iterations can be adjusted so that it will have an arbitrarily large computation time. However, its computation requires very little RAM, which makes it susceptible to brute-force attacks from dedicated hardware like GPUs and ASICs.

Scrypt was designed to require large amounts of memory and mitigate attacks from dedicated hardware, which makes it a much stronger choice than PBKDF2. However, scrypt's data-dependent memory accesses can be susceptible to side-channel attacks.

Therefore, we feel that `argon2id` is a much better alternative as it offers protections against both side channels and dedicated hardware. This stackexchange answer provides excellent insight as to why, and this one explains the parameters nicely. Also, `argon2` was the winner of the 2015 Password Hashing Competition.

**Recommendations**
Short term, consider revising EIP-2335 to include `argon2id` as a possible KDF. As argued, `argon2id` is a better choice than scrypt or PBKDF2. Incorporating `argon2id` into EIP-2335 will provide better security to the community as a whole.

Long term, if EIP-2335 is revised, implement the use of `argon2id` as a KDF within `eth2_keystore`. This will provide better security to the derived keys.

**References**
- Open Sesame: The Password Hashing Competition and Argon2

## 12. Bias in BLS secret key generation

Severity: Low
Type: Cryptography
Target: `eth2_key_derivation`, `EIP-2333`

Difficulty: High
Finding ID: TOB-LIHO-012

**Description**
BLS secret keys need to be generated uniformly randomly to be secure. Currently, BLS secret keys are generated by computing an HKDF over a seed (an array of bytes) and then reducing the result modulo the group order. However, generating keys in this manner is not uniform; specifically, smaller secret keys will be generated more often. To see this more easily, consider the possible results of taking a 3-bit integer modulo 3. 0 occurs 3 times (0, 3, 6), while 2 only occurs 2 times (2, 5).

In general, the security of the BLS signature scheme is proven under the assumption that secret keys are generated uniformly randomly. Therefore, we cannot make any of the same security guarantees if keys are not generated properly. In addition, slight bias has a history of dramatically reducing the security bounds of any cryptosystem.

**Recommendations**
Short term, since adjusting the EIP will provide better security to the community as a whole, consider revising EIP-2333 so that the `mod_r` operation becomes uniformly distributed, or can fail. The latter is easily implemented by requiring the number to lie within a range that is uniformly distributed after a `mod_r` operation. For instance, with 3-bit integers mod 3, we only accept HKDF results that lie in the range 0-5, as 6 and 7 do not cover the full range of 0, 1, 2. For these results, we can error out or just take bits 2-4 instead of 1-3, and keep going until we get something usable.

Long term, implement any changes accepted to EIP-2333 to reduce the risk of a secret key compromise.

**References**
- LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage

## 13. Unnecessary use of panicking functions

Severity: Informational
Type: Denial of Service
Target: `beacon_node/beacon_chain/src/beacon_chain.rs`, `beacon_node/network/src/sync/manager.rs`

Difficulty: Not applicable
Finding ID: TOB-LIHO-013

**Description**
There are a few places in the codebase that use functions such as `panic!`, `expect`, or `unwrap`. When executed, these functions will likely cause the program to terminate. Currently, the way they are used is not dangerous: The preconditions are met, so termination doesn't occur. In some places, the code can be rewritten in such a way that the functions are not required—see the examples in Figures 13.1 and 13.2.

```
while !filtered_chain_segment.is_empty() {
    // Determine the epoch of the first block in the remaining segment.
    let start_epoch = filtered_chain_segment
        .first()
        .map(|(_root, block)| block)
        .expect("chain_segment cannot be empty")
        .slot()
        .epoch(T::EthSpec::slots_per_epoch());
    ...
}
```

*Figure 13.1:* beacon_node/beacon_chain/src/beacon_chain.rs#L1231-L1238.

```
while let Some((_root, block)) = filtered_chain_segment.first() {
    // Determine the epoch of the first block in the remaining segment.
    let start_epoch = block
        .slot()
        .epoch(T::EthSpec::slots_per_epoch());
    ...
}
```

*Figure 13.2: Code from Figure 13.1 rewritten without* `expect()`.

A similar example concerns the `process_parent_request` function in `beacon_node/network/src/sync/manager.rs`.

**Exploit Scenario**
A Lighthouse developer moves code calling the `expect` function without taking preconditions into account, potentially forcing a node to crash.

**Recommendations**

Short term, change the code to eliminate panics in the indicated places. This will eliminate a potential denial-of-service vector.

Long term, think about further minimizing panic occurrences throughout the codebase. Also, consider using `no-panic` to reduce the likelihood of a panic becoming reachable in future revisions.

## 14. Password to validator private key is stored in plaintext

Severity: High
Type: Data Exposure
Target: Validator

Difficulty: High
Finding ID: TOB-LIHO-014

**Description**
Validator private keys are stored on disk and encrypted symmetrically with AES. The AES key is also stored on disk in a separate directory in plaintext. The encryption provides little security since the decryption key can be found just in a separate directory. By default, the directory for storing the decryption keys is `~/.lighthouse/secrets` and has to be manually created by the user before running Lighthouse. Leaning on the user to make sure the directory permissions are set correctly can weaken security further.

**Exploit Scenario**
An attacker gains access to the filesystem of a machine running Lighthouse and is able to steal all validator private keys.

**Recommendations**
Short term, encrypt the validator key with a password and require the user to enter the password on Lighthouse startup. This will eliminate the need to store sensitive information in plaintext on the filesystem.

Long term, give users the option of storing their credentials in an HSM (Hardware Security Module) or password manager such as Vault. This will allow the wallet to offload responsibility for sensitive material to a service designed to handle such information.

## 15. Password to wallet is stored in plaintext

Severity: High
Type: Data Exposure
Target: Lighthouse wallet

Difficulty: High
Finding ID: TOB-LIHO-015

**Description**
Creating a wallet requires a password file (Figure 14.1). Because the password is needed later to create validator key pairs, it is impossible to create a wallet without storing the password in file.

```
$ lighthouse account wallet create --name foo
error: The following required arguments were not provided:
    --passphrase-file <WALLET_PASSWORD_PATH>
...
```

*Figure 14.1: A passphrase file is required for a wallet to be created.*

**Exploit Scenario**
An attacker gains access to the filesystem of a machine running Lighthouse and is able to steal the wallet.

**Recommendations**
Short term, require the user to enter the wallet password on Lighthouse startup. This will eliminate the need to store sensitive information in plaintext on the filesystem.

Long term, give users the option of storing their credentials in an HSM (Hardware Security Module) or password manager such as Vault. This will allow the wallet to offload responsibility for sensitive material to a service designed to handle such information.

## 16. Secret key passed as CLI argument

Severity: High
Type: Data Exposure
Target: beacon node

Difficulty: High
Finding ID: TOB-LIHO-016

**Description**
Beacon node allows a secret key to be passed as a CLI argument to the process. It is not recommended to pass sensitive information through a command line since it can be easily leaked to other users of the system. Reporting frameworks might also capture and export it to a remote location. Additionally, a whole command along with the secret might be saved in shell history. The relevant switch is presented in Figure 16.1.

```
$ lighthouse beacon_node --help
...
--p2p-priv-key <HEX>
            A secp256k1 secret key, represented as ASCII-encoded hex bytes (with or
without 0x prefix). Default is
            either loaded from disk or generated automatically.
...
```

*Figure 16.1: Passing a secret through the command line.*

**Exploit Scenario**
An attacker gains access to a machine running Lighthouse and is able to read the `p2p-priv-key` through `cat /proc/PID/cmdline`.

**Recommendations**
Short term, remove the option to pass `p2p-priv-key` from the command line. Passing this option on the command line makes it accessible to an attacker.

Long term, as new command line options are added to the system, ensure that they do not involve sensitive information. Command line options are generally accessible to an attacker and could reveal such information.

## 17. Insufficient network layer unit tests

Severity: Informational
Type: Error Reporting
Target: `eth2_libp2p` and `network` crates

Difficulty: Undetermined
Finding ID: TOB-LIHO-017

**Description**
The `eth2_libp2p` and `network` could benefit from additional unit tests. For example, many `network` crate functions are not currently exercised by any unit test (Tables 17.1 and 17.2).

| Directory | Line Coverage | Function Coverage |
|---|---|---|
| `src` | 47.5 % | 21.2 % |
| `src/behaviour` | 65.5 % | 56.4 % |
| `src/behaviour/handler` | 51.3 % | 54.2 % |
| `src/discovery` | 40.3 % | 25.0 % |
| `src/peer_manager` | 64.1 % | 47.4 % |
| `src/rpc` | 64.2 % | 25.0 % |
| `src/rpc/codec` | 72.4 % | 79.4 % |
| `src/types` | 15.0 % | 6.8 % |

*Table 17.1:* `eth2_libp2p` *test code coverage as reported by* `grcov`*.*

| Directory | Line Coverage | Function Coverage |
|---|---|---|
| `src` | 24.4 % | 25.0 % |
| `src/attestation_service` | 74.5 % | 43.9 % |
| `src/beacon_processor` | 0.0 % | 0.0 % |
| `src/sync` | 0.0 % | 0.0 % |
| `src/sync/range_sync` | 0.0 % | 0.0 % |

*Table 17.2:* `network` *test code coverage as reported by* `grcov`*.*

Unit tests help expose errors, provide a sort of documentation of the code, and can be used to generate fuzzing corpora (see Appendix D). Moreover, unit tests exercise code in a more systematic way than any human can, and thus help protect against regressions.

**Exploit Scenario**
Eve exploits a flaw in an `eth2_libp2p` or `network` crate function. The flaw would likely have been revealed through unit tests.

**Recommendations**
Short term, add unit tests for `eth2_libp2p` and `network` crate functions not currently exercised by unit tests. Ideally, there will be at least one test for each "happy" (successful) path, and at least one test for each "sad" (failing) path. A comprehensive set of unit tests will help expose errors, protect against regressions, and provide a sort of documentation to users.

Long term, ensure unit tests are written for new functions as they are added to the codebase to prevent them from introducing bugs.

## 18. Memory exhaustion via `Multiaddr` deserialization

Severity: Low
Type: Denial of Service
Target: `beacon_node/network/src/nat.rs`, `beacon_node/eth2_libp2p/src/peer_manager/client.rs`

Difficulty: High
Finding ID: TOB-LIHO-018

**Description**
When deserializing a `Multiaddr`, Lighthouse's `rust-libp2p` dependency can allocate vectors of arbitrary size. This behavior could be exploited for denial of service.

The vulnerable function is `visit_seq` in Figure 18.1. The function passes `seq.size_hint()` (which can be arbitrarily large) to `Vec::with_capacity`. For comparison, `serde`'s code for deserializing a sequence appears in Figures 18.2 and 18.3. Note that `serde`'s implementation limits the size of the vector to 4096.

```
impl<'de> Deserialize<'de> for Multiaddr {
    fn deserialize<D>(deserializer: D) -> StdResult<Self, D::Error>
    where
        D: Deserializer<'de>,
    {
        struct Visitor { is_human_readable: bool };

        impl<'de> de::Visitor<'de> for Visitor {
            type Value = Multiaddr;
            ...
            fn visit_seq<A: de::SeqAccess<'de>>(self, mut seq: A) ->
StdResult<Self::Value, A::Error> {
                let mut buf: Vec<u8> = Vec::with_capacity(seq.size_hint().unwrap_or(0));
```

*Figure 18.1:* rust-libp2p/misc/multiaddr/src/lib.rs#L335-L349.

```
            fn visit_seq<A>(self, mut seq: A) -> Result<Self::Value, A::Error>
            where
                A: SeqAccess<'de>,
            {
                let mut values = Vec::with_capacity(size_hint::cautious(seq.size_hint()));
```

*Figure 18.2:* serde/src/de/impls.rs#L863-L867.

```
    #[inline]
    pub fn cautious(hint: Option<usize>) -> usize {
        cmp::min(hint.unwrap_or(0), 4096)
    }
```

*Figure 18.3:* serde/src/private/de.rs#L201-L204.

This bug was found indirectly by fuzzing `UPnPConfig`'s `From` trait implementations. Specifically, `NetworkConfig`'s deserialization code was used to prepare fuzzing inputs for:

```
fn from(config: &NetworkConfig) -> UPnPConfig
```

A stack trace appears in Figure 18.4.

```
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  __GI_abort () at abort.c:79
#2  std::sys::unix::abort_internal () at library/std/src/sys/unix/mod.rs:231
#3  std::process::abort () at library/std/src/process.rs:1773
#4  std::alloc::rust_oom () at library/std/src/alloc.rs:331
#5  alloc::alloc::__default_lib_allocator::__rg_oom () at library/alloc/src/alloc.rs:389
#6  alloc::alloc::handle_alloc_error () at library/alloc/src/alloc.rs:343
#7  alloc::raw_vec::RawVec<T,A>::allocate_in ()
#8  <<parity_multiaddr::Multiaddr as serde::de::Deserialize>::deserialize::Visitor as
serde::de::Visitor>::visit_seq ()
#9  serde_cbor::de::Deserializer<R>::parse_array ()
#10 serde_cbor::de::Deserializer<R>::parse_value ()
#11 <parity_multiaddr::Multiaddr as serde::de::Deserialize>::deserialize ()
#12 <serde_cbor::de::SeqAccess<R> as serde::de::SeqAccess>::next_element_seed ()
#13 <serde::de::impls::<impl serde::de::Deserialize for
alloc::vec::Vec<T>>::deserialize::VecVisitor<T> as serde::de::Visitor>::visit_seq ()
#14 serde_cbor::de::Deserializer<R>::parse_array ()
#15 serde_cbor::de::Deserializer<R>::parse_value ()
#16 serde::de::impls::<impl serde::de::Deserialize for alloc::vec::Vec<T>>::deserialize ()
#17 serde_cbor::de::Deserializer<R>::parse_map ()
#18 serde_cbor::de::Deserializer<R>::parse_value ()
#19 eth2_libp2p::config::_::<impl serde::de::Deserialize for
eth2_libp2p::config::Config>::deserialize ()
```

*Figure 18.4: Stack trace leading to the bug in Figure 18.1.*

Note that `IdentifyInfo` (another `rust-libp2p` type used by Lighthouse) also uses `Multiaddr`. If `rust-libp2p` were to derive the `serde::Deserialize` trait for that type, the resulting code would be vulnerable as well.

**Exploit Scenario**
Eve convinces Alice to use her network configuration. Alice's node crashes as a result. Alice wastes time and effort trying to identify the cause of the crash.

**Recommendations**
Short term, patch the code in Figure 18.1 by limiting the size of the value passed to `Vec::with_capacity`. Use the patched version of `rust-libp2p` until the bug is fixed upstream. These steps will eliminate a potential denial-of-service attack.

Long term, consider adding the fuzz targets in Appendix D to the set of targets that you regularly fuzz. This bug was found by fuzzing one of those targets, so regularly fuzzing them could help to identify similar bugs, especially as Lighthouse's dependencies evolve.

## 19. SSZ snappy decoder reads more data than specification recommends

Severity: Informational | Difficulty: High
Type: Data Validation | Finding ID: TOB-LIHO-019
Target: `beacon_node/eth2_libp2p/src/rpc/codec/ssz_snappy.rs`

**Description**
The Ethereum 2.0 specification recommends an upper bound for the amount of compressed data that a node should read when processing RPC messages. Lighthouse deviates from the specification by checking whether the bound was exceeded *after* the data has been read.

The passage in question is the following (SSZ-snappy encoding strategy):

> A reader SHOULD NOT read more than `max_encoded_len(n)` bytes after reading the SSZ length-prefix `n` from the header.

However, the Lighthouse implementation reads as much data as will fit in a provided buffer, and checks after the fact whether the bound was exceeded (Figures 19.1 and 19.2).

```
        // Should not attempt to decode rpc chunks with length > max_packet_size
        if length > self.max_packet_size {
            return Err(RPCError::InvalidData);
        }
        let mut reader = FrameDecoder::new(Cursor::new(&src));
        let mut decoded_buffer = vec![0; length];

        match read_exact(&mut reader, &mut decoded_buffer, length) {
```

*Figure 19.1:* beacon_node/eth2_libp2p/src/rpc/codec/ssz_snappy.rs#L113-L120.

```
fn read_exact<T: std::convert::AsRef<[u8]>>(
    reader: &mut FrameDecoder<Cursor<T>>,
    mut buf: &mut [u8],
    uncompressed_length: usize,
) -> Result<(), std::io::Error> {
    // Calculate worst case compression length for given uncompressed length
    let max_compressed_len = snap::raw::max_compress_len(uncompressed_length) as u64;

    // Initialize the position of the reader
    let mut pos = reader.get_ref().position();
    let mut count = 0;
    while !buf.is_empty() {
        match reader.read(buf) {
            ...
        }
        // Get current position of reader
```

```
        let curr_pos = reader.get_ref().position();
        ...
        if curr_pos > pos {
            count += reader.get_ref().position() - pos;
            pos = curr_pos;
        } else {
            ...
        }

        if count > max_compressed_len {
            return Err(std::io::Error::new(
                ErrorKind::InvalidData,
                "snappy: compressed data is > max_compressed_len",
            ));
        }
    }
```

*Figure 19.2:* beacon_node/eth2_libp2p/src/rpc/codec/ssz_snappy.rs#L424-L466.

Note that `buf` in Figure 19.2 is a buffer at most `MAX_RPC_SIZE` ($2^{10}$) bytes in size. Thus, it appears that in the worst case, this deviation could cause a node to read about one megabyte of traffic unnecessarily.

**Exploit Scenario**
Alice runs a Lighthouse node. Eve repeatedly sends Alice's node traffic that does not decompress correctly. This combined with other factors makes Alice's node unable to keep up with the network.

**Recommendations**
Short term, when reading SSZ encoded data, limit the size of the read buffer to `max_compressed_len`. This will make a subsequent check against `max_compressed_len` unnecessary and will bring Lighthouse more in line with the Ethereum 2.0 specification.

Long term, adhere to the specification unless there is a good reason not to do so. This will help Lighthouse avoid problems already anticipated by the specification's authors.

## 20. `Gossipsub` parameters deviate from the specification

Severity: Informational
Type: Configuration
Target: `beacon_node/eth2_libp2p/src/config.rs`

Difficulty: Not applicable
Finding ID: TOB-LIHO-020

**Description**
The Ethereum 2.0 networking specification states that clients must support the `gossipsub v1 libp2p` protocol for broadcasting topics. `gossipsub` was proposed to improve on the initial implementation, which used `floodsub`, a simple propagation strategy in which nodes "flood" the network by broadcasting to every node they know about. This was problematic because bandwidth costs were excessive for highly connected peers, decreasing scalability.

`gossipsub` was proposed to address the shortcomings of `floodsub` (as discussed in the gossipsub specification) and constrain the outdegree of nodes in the network to achieve better performance. The topology of this network is then determined by a series of parameters that define, for example, the desired outbound degree and the maximum outbound degree.

Among other parameters, the Ethereum 2.0 networking specification assigns values to `D`, `D_low`, and `D_high`, which respectively define the desired outbound degree, minimum outbound degree, and maximum outbound degree for nodes in the network. Per the specification:

`D` = 6
`D_low` = 5
`D_high` = 12

However, the implementation uses different values (see Figure 20.1). In particular, the values for `D` and `D_low` are different in the implementation.

```
        // gossipsub configuration
        // Note: The topics by default are sent as plain strings. Hashes are an optional
        // parameter.
        let gs_config = GossipsubConfigBuilder::new()
            .max_transmit_size(GOSSIP_MAX_SIZE)
            .heartbeat_interval(Duration::from_millis(700))
            .mesh_n(8)
            .mesh_n_low(6)
            .mesh_n_high(12)
            .gossip_lazy(6)
            .fanout_ttl(Duration::from_secs(60))
            .history_length(6)
            .history_gossip(3)
            .validate_messages() // require validation before propagation
            .validation_mode(ValidationMode::Anonymous)
            // prevent duplicates for 550 heartbeats(700millis * 550) = 385 secs
```

```
            .duplicate_cache_time(Duration::from_secs(385))
            .message_id_fn(gossip_message_id)
            .build()
            .expect("valid gossipsub configuration");
```

*Figure 20.1:* beacon_node/eth2_libp2p/src/config.rs#L97-L116.

Since these configuration parameters dictate the topology of the network, it is important to determine the optimal parameters for the network and ensure these parameters are used consistently across the network. Therefore, it is imperative that the implementation and specification use the same parameters.

**Recommendations**

Short term, adjust either the implementation or the specification so that the parameter choices match and desired network performance is achieved.

Long term, if changes are made to any of these parameters, change both the implementation and specification simultaneously so they maintain parity as they each evolve.

# A. Vulnerability Classifications

| Vulnerability Classes | |
|---|---|
| **Class** | **Description** |
| Access Controls | Related to authorization of users and assessment of rights |
| Auditing and Logging | Related to auditing of actions or logging of problems |
| Authentication | Related to the identification of users |
| Configuration | Related to security configurations of servers, devices, or software |
| Cryptography | Related to protecting the privacy or integrity of data |
| Data Exposure | Related to unintended exposure of sensitive information |
| Data Validation | Related to improper reliance on the structure or values of data |
| Denial of Service | Related to causing system failure |
| Error Reporting | Related to the reporting of error conditions in a secure fashion |
| Patching | Related to keeping software up to date |
| Session Management | Related to the identification of authenticated users |
| Timing | Related to race conditions, locking, or order of operations |
| Undefined Behavior | Related to undefined behavior triggered by the program |

| Severity Categories | |
|---|---|
| **Severity** | **Description** |
| Informational | The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth |
| Undetermined | The extent of the risk was not determined during this engagement |
| Low | The risk is relatively small or is not a risk the customer has indicated is important |
| Medium | Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal |

| | |
|---|---|
| | implications for client |
| High | Large numbers of users, very bad for client's reputation, or serious legal or financial implications |

| **Difficulty Levels** | |
|---|---|
| **Difficulty** | **Description** |
| Undetermined | The difficulty of exploit was not determined during this engagement |
| Low | Commonly exploited, public tools exist or can be scripted that exploit this flaw |
| Medium | Attackers must write an exploit, or need an in-depth knowledge of a complex system |
| High | The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue |

# B. Non-Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

- **Keyword "`mut`" is used unnecessarily.** Within the body of `per_block_processing`, the object referred to by `state` is mutated, but `state` itself is not. Thus, the uses of "`mut`" in Figures B.1 and B.2 can be eliminated.

```
pub fn per_block_processing<T: EthSpec>(
   mut state: &mut BeaconState<T>,
   signed_block: &SignedBeaconBlock<T>,
   block_root: Option<Hash256>,
   block_signature_strategy: BlockSignatureStrategy,
   spec: &ChainSpec,
) -> Result<(), BlockProcessingError> {
```

*Figure B.1:* consensus/state_processing/src/per_block_processing.rs#L80-L86.

```
   process_randao(&mut state, &block, verify_signatures, &spec)?;
   process_eth1_data(&mut state, &block.body.eth1_data)?;
   process_proposer_slashings(
       &mut state,
       &block.body.proposer_slashings,
       verify_signatures,
       spec,
   )?;
```

*Figure B.2:* consensus/state_processing/src/per_block_processing.rs#L118-L125.

- **Occurrences of the `unwrap` function.** These can at least be changed to `expect`, stating the invariant that is assumed to be held, thereby providing better documentation and crash messages.

```
while self.n_dc > MAX_DC_PEERS {
    let to_drop = self
        .peers
        .iter()
        .filter(|(_, info)| info.connection_status.is_disconnected())
        .min_by_key(|(_, info)| info.reputation)
        .map(|(id, _)| id.clone())
        .unwrap(); // should be safe since n_dc > MAX_DC_PEERS > 0
```

*Figure B.3:* beacon_node/eth2-libp2p/src/peer_manager/peerdb.rs#L281-L288.

- **The following code is unused:**

```
impl<T> ValidatorDutyBase<T> {
    /// Given a `slot_signature` determines if the validator of this duty is an
aggregator.
    // Note that we assume the signature is for the associated pubkey to avoid the
signature
    // verification
    pub fn is_aggregator(&self, slot_signature: &Signature) -> bool {
        if let Some(modulo) = self.aggregator_modulo {
            let signature_hash = hash(&slot_signature.as_bytes());
            let signature_hash_int = u64::from_le_bytes(
                signature_hash[0..8]
                    .try_into()
                    .expect("first 8 bytes of signature should always convert to fixed
array"),
            );
            signature_hash_int % modulo == 0
        } else {
            false
        }
    }
}
```

*Figure B.4:* common/rest_types/src/validator.rs#L33-L50.

- **`Behaviour` field initialization does not match the order in which the fields are declared (Figures B.5 and B.6).** Consider making the fields' declaration and initialization orders consistent.

```
pub struct Behaviour<TSpec: EthSpec> {
    /// The routing pub-sub mechanism for eth2.
    gossipsub: Gossipsub,
    /// The Eth2 RPC specified in the wire-0 protocol.
    eth2_rpc: RPC<TSpec>,
```

*Figure B.5:* beacon_node/eth2_libp2p/src/behaviour/mod.rs#L99-L103.

```
Ok(Behaviour {
     eth2_rpc: RPC::new(log.clone()),
     gossipsub,
```

*Figure B.6:* beacon_node/eth2_libp2p/src/behaviour/mod.rs#L159-L161.

- **Variables `E`, `T`, `TSpec`, and `TEthSpec` are used to represent structs that implement the `EthSpec` trait.** Examples:
    - eth2_libp2p/tests/common/mod.rs#L12: `type E = MinimalEthSpec;`

- eth2_libp2p/src/rpc/mod.rs#L41: `pub enum RPCSend<T: EthSpec> {`
- eth2_libp2p/src/behaviour/mod.rs#L48: `pub enum BehaviourEvent<TSpec: EthSpec> {`
- beacon_chain/src/builder.rs#L77: `TEthSpec: EthSpec + 'static,`

Since `E` and `T` are rather generic, consider using `TSpec` or `TEthSpec` consistently.

- **In `From<u64>` implementation for `GoodbyeReason`, the return type could be changed to `Self` (Figure B.7).**

```
impl From<u64> for GoodbyeReason {
    fn from(id: u64) -> GoodbyeReason {
```

*Figure B.7:* beacon_node/eth2_libp2p/src/rpc/methods.rs#L135-L136.

- **`SyncInfo` field names contain unnecessary prefixes (Figure B.8).** Consider eliminating the "`status_`" prefixes.

```
pub struct SyncInfo {
    pub status_head_slot: Slot,
    pub status_head_root: Hash256,
    pub status_finalized_epoch: Epoch,
    pub status_finalized_root: Hash256,
}
```

*Figure B.8:* beacon_node/eth2_libp2p/src/peer_manager/peer_sync_status.rs#L22-L27.

# C. Fix Log

On August 24 and December 14-18, 2020, Trail of Bits reviewed Sigma Prime's fixes for the issues identified in this report. The fixes were spread across the following 16 pull requests, all of which were merged.

- https://github.com/sigp/lighthouse/pull/1181
- https://github.com/sigp/lighthouse/pull/1192
- https://github.com/sigp/lighthouse/pull/1195
- https://github.com/sigp/lighthouse/pull/1210
- https://github.com/sigp/lighthouse/pull/1227
- https://github.com/sigp/lighthouse/pull/1270
- https://github.com/sigp/lighthouse/pull/1277
- https://github.com/sigp/lighthouse/pull/1278
- https://github.com/sigp/lighthouse/pull/1322
- https://github.com/sigp/lighthouse/pull/1327
- https://github.com/sigp/lighthouse/pull/1330
- https://github.com/sigp/lighthouse/pull/1334
- https://github.com/sigp/lighthouse/pull/1738
- https://github.com/sigp/lighthouse/pull/1867
- https://github.com/libp2p/rust-libp2p/pull/1833
- https://github.com/ethereum/eth2.0-specs/pull/2121

Sigma Prime fixed or partially fixed 13 of the 20 findings identified in this report. We reviewed the fixes to ensure that they would be effective. Of the remaining seven findings, Sigma Prime chose to accept the risk associated with five of them, and has not yet fixed two of them.

| ID | Title | Severity | Status |
|---|---|---|---|
| 01 | Codebase uses a crate with a RUSTSEC advisory | Medium | Fixed |
| 02 | Build process relies on outdated dependencies | Informational | Fixed |
| 03 | Assumptions about struct field initialization order | Low | Fixed |
| 04 | Comments suggest code and documentation are out of date | Informational | Fixed |
| 05 | Downloaded deposit contract is not validated with a checksum | Informational | Fixed |
| 06 | `BeaconState` objects are mutated upon error | Informational | Risk |

| | | | |
|---|---|---|---|
| | | | Accepted |
| 07 | Errors produced by "`ParallelValidatorTreeHash::leaves`" are non-deterministic | Informational | Risk Accepted |
| 08 | Memory leak due to non-graceful shutdown | Low | Fixed |
| 09 | Builder pattern is not strictly adhered to | Informational | Risk Accepted |
| 10 | `rust-crypto` is unmaintained and a better alternative should be used | Informational | Fixed |
| 11 | Consider using `argon2id` as a KDF | Informational | Not Fixed |
| 12 | Bias in BLS secret key generation | Low | Risk Accepted |
| 13 | Unnecessary use of panicking functions | Informational | Fixed |
| 14 | Password to validator private key is stored in plaintext | High | Risk Accepted |
| 15 | Password to wallet is stored in plaintext | High | Not Fixed |
| 16 | Secret key passed as CLI argument | High | Fixed |
| 17 | Insufficient network layer unit tests | Informational | Partially Fixed |
| 18 | Memory exhaustion via Multiaddr deserialization | Low | Fixed |
| 19 | SSZ snappy decoder reads more data than specification recommends | Informational | Fixed |
| 20 | Gossipsub parameters deviate from the specification | Informational | Fixed |

## Detailed Fix Log

**Finding 1: Codebase uses a crate with a RUSTSEC advisory**
Fixed. Lighthouse now uses `rusqlite 0.23.1`.

**Finding 2: Build process relies on outdated dependencies**
Fixed. The `db-key` dependency could not be upgraded because `leveldb` (another dependency) relies on the version currently used. According to a comment from the `db-key` author, the newer version is "historic" and related to a change that "didn't yet work out." All other cited dependencies were upgraded to the recommended version or newer.

**Finding 3: Assumptions about struct field initialization order**
Fixed. The `Decode` macro was adjusted so that the deserialized values are first stored in local variables, which are then used to initialize the derived struct's fields. This eliminates assumptions about the order in which those fields are initialized.

**Finding 4: Comments suggest code and documentation are out of date**
Fixed. The cited examples were fixed.

**Finding 5: Downloaded deposit contract is not validated with a checksum**
Fixed. The `sha256` hashes of both the deposit contract ABI and its bytecode are now checked against known constants.

**Finding 6: `BeaconState` objects are mutated upon error**
Risk accepted.

**Finding 7: Errors produced by "`ParallelValidatorTreeHash::leaves`" are non-deterministic**
Risk accepted.

**Finding 8: Memory leak due to non-graceful shutdown**
Fixed. The `tokio` tasks now wait for an "exit" signal that is triggered in various places in the code. Running the test in Figure 8.2 no longer triggers the warning.

**Finding 9: Builder pattern is not strictly adhered to**
Risk accepted.

**Finding 10: `rust-crypto` is unmaintained and a better alternative should be used**
Fixed. Functionality previously provided by `rust-crypto` is now provided by the following crates:

- `aes-ctr`

- `hmac`
- `pbkdf2`
- `scrypt`
- `sha2`

**Finding 11: Consider using `argon2id` as a KDF**
Not fixed.

**Finding 12: Bias in BLS secret key generation**
Risk accepted.

**Finding 13: Unnecessary use of panicking functions**
Fixed. The cited example was rewritten to eliminate `expect`.

**Finding 14: Password to validator private key is stored in plaintext**
Risk accepted.

**Finding 15: Password to wallet is stored in plaintext**
Not fixed.

**Finding 16: Secret key passed as CLI argument**
Fixed. The CLI argument was eliminated. The key is now loaded from a file within the "network" directory, which is configurable.

**Finding 17: Insufficient network layer unit tests**
Partially fixed. The finding was addressed by commit `e477390` (PR #1867). The code coverage in commit `e477390` does not appear to have changed significantly from Tables 17.1 and 17.2. However, the number of `eth2_libp2p` tests did increase:

| Package | Commit `da44821` | Commit `e477390` |
|---|---|---|
| `eth2_libp2p` | 16 | 25 |
| `network` | 2 | 2 |

*Number of* `eth2_libp2p` *and* `network` *tests in commits* `da44821` *and* `e477390`*.*

**Finding 18: Memory exhaustion via `Multiaddr` deserialization**
Fixed. The "size hint" is now bounded to 4096. Note: we verified that the fix appears not only in `rust-libp2p` upstream, but also in the patched version that Lighthouse uses.

**Finding 19: SSZ snappy decoder reads more data than specification recommends**
Fixed. The implementation now uses a `Take` reader to limit the amount of data that is read.

**Finding 20: `Gossipsub` parameters deviate from the specification**
Fixed. The specification was updated to match the Lighthouse implementation (as opposed to the other way around).

## Detailed Issue Discussion

Responses from Sigma Prime for each issue are included as quotes below.

**Finding 1: Codebase uses a crate with a RUSTSEC advisory**

> *Package upgraded: https://github.com/sigp/lighthouse/issues/1194*
> *Cargo Audit added to CI: https://github.com/sigp/lighthouse/pull/1192*

**Finding 2: Build process relies on outdated dependencies**

> *Fixed in: https://github.com/sigp/lighthouse/pull/1322*

**Finding 3: Assumptions about struct field initialization order**

> *Fixed in: https://github.com/sigp/lighthouse/pull/1210*

**Finding 4: Comments suggest code and documentation are out of date**

> *Fixed in:*
> - *https://github.com/sigp/lighthouse/pull/1227*
> - *https://github.com/sigp/lighthouse/pull/1327*
> - *https://github.com/sigp/lighthouse/pull/1334*

**Finding 5: Downloaded deposit contract is not validated with a checksum**

> *Fixed in: https://github.com/sigp/lighthouse/pull/1330*

**Finding 6: `BeaconState` objects are mutated upon error**

> *We are aware that states are mutated; however they are always discarded on an error. This is how the specification is written, and to deviate would make it much more difficult to compare our implementation to the specification.*
>
> *In our opinion, the cons of implementing this suggestion would outweigh the pros. As such, we respectfully choose to not implement this suggestion, but thank Trail of Bits for raising it.*

**Finding 7: Errors produced by "`ParallelValidatorTreeHash::leaves`" are non-deterministic**

*This suggestion becomes a trade-off between exiting early with one, non-deterministic error or exiting late with all errors.*

*Given that we do not allow any errors to be returned from this function, we find it more valuable to exit as soon as possible, rather than do the additional computation to collect all errors that may occur.*

*We respectfully choose to not implement this suggestion, but we thank Trail of Bits for raising it.*

**Finding 8: Memory leak due to non-graceful shutdown**

*Fixed in: https://github.com/sigp/lighthouse/pull/1181/commits/05766811d8c26c908c10723d1f7a9d887cc5ba1a*

**Finding 9: Builder pattern is not strictly adhered to**

*We believe that it is reasonable to choose between the builder and init-struct patterns across a project. The builder pattern allows for elegant type-inference at compile-time, whilst the init-struct provides simple runtime configuration for a config file. They are different tools to be used for different jobs.*

*The* `ProductionBeaconNode` *is always initialized at runtime with static types from a config file. On the other hand, the* `ClientBuilder` *uses the builder pattern so it can be constructed (at compile time) using different types.*

*We will review our use of struct instantiation code and consider how we can make it more elegant.*

**Finding 10: `rust-crypto` is unmaintained and a better alternative should be used**

*Fixed in: https://github.com/sigp/lighthouse/pull/1270*

**Finding 11: Consider using `argon2id` as a KDF**

*We have raised this with the author of EIP-2335 and await a decision.*

**Finding 12: Bias in BLS secret key generation**

*EIP-2333 specifically mentions this bias and deems it to be acceptable.*

**Finding 13: Unnecessary use of panicking functions**

*Fixed in https://github.com/sigp/lighthouse/pull/1278*

**Finding 14: Password to validator private key is stored in plaintext**

*Unfortunately, here we must choose between security and usability. If we were to rely upon user input to decrypt validator keys then the validator client would not be able to reboot without user input. This is undesirable since we wish for the validator client to act as a typical server application which rarely, if ever, requires user input.*

*The password files are created with strict file permissions and we understand that we are following best practices (assuming no mandatory user input).*

*We look forward to hardware wallets that may help us solve this problem with a higher degree of security, but these do not yet exist.*

*We respectfully choose to not implement this suggestion, but we thank Trail of Bits for raising it.*

**Finding 15: Password to wallet is stored in plaintext**

*We plan to implement more user-friendly methods for account management in upcoming releases. The current functionality is geared towards automated deployment (e.g., Ansible). We acknowledge this issue and plan to address it in a future release.*

**Finding 16: Secret key passed as CLI argument**

*Whilst this was a temporary solution for testing, we have fixed it in: https://github.com/sigp/lighthouse/pull/1277*

**Finding 17: Insufficient network layer unit tests**

*Further tests have been added however complete code coverage of this area is very difficult. We have in-built simulations that are used to test the complex logic here on every code change. Further tests have been added in: https://github.com/sigp/lighthouse/pull/1867*

**Finding 18: Memory exhaustion via `Multiaddr` deserialization**

*This has been resolved in: https://github.com/libp2p/rust-libp2p/pull/1833*

**Finding 19: SSZ snappy decoder reads more data than specification recommends**

*This has been fixed in: https://github.com/sigp/lighthouse/pull/1738*

**Finding 20: `Gossipsub` parameters deviate from the specification**

*The spec has been updated in this PR: https://github.com/ethereum/eth2.0-specs/pull/2121*

## D. `From` Trait Implementation Fuzzing

During the network phase of the engagement, Trail of Bits fuzzed all but two of the `eth2_libp2p` and `network` crates' `From` trait implementations, as well as some related functions.

Table D.1 lists the 18 functions that were fuzzed. We fuzzed these using `cargo afl`. Each fuzz target was given its own core, and was fuzzed for at least six hours. If a function was called in a unit test, we used the arguments to seed the corpus. Otherwise, we used the argument type's default values to seed the corpus.

| Function | Non-default corpus? | Crash found? |
|---|---|---|
| `fn from(req: Request) -> RPCRequest<TSpec>` | No | No |
| `fn from(resp: Response<TSpec>) -> RPCCodedResponse<TSpec>` | No | No |
| `fn from_identify_info(info: &IdentifyInfo) -> Client` | No | **Yes** |
| `fn from(f: f64) -> Score` | No | No |
| `fn from(s: String) -> ErrorType` | No | No |
| `fn from(s: &str) -> ErrorType` | No | No |
| `fn from(id: u64) -> GoodbyeReason` | No | No |
| `fn from_ssz_bytes(bytes: &[u8]) -> Result<GoodbyeReason, ssz::DecodeError>` | No | No |
| `pub fn from_error(response_code: u8, err: ErrorType) -> RPCCodedResponse::<T>` | No | No |
| `fn from(err: ssz::DecodeError) -> RPCError` | No | No |
| `fn from_quota(quota: Quota) -> Result<Limiter::<Key>, &'static str>` | No | No |
| `fn decode(topic: &str) -> Result<GossipTopic, String>` | **Yes** | No |
| `fn from(subnet_id: SubnetId) -> GossipKind` | No | No |
| `fn from(peer_id: PeerIdSerialized) ->` | No | No |

| | | |
|---|---|---|
| `PeerId` | | |
| `fn from_str(s: &str) -> Result<PeerIdSerialized, String>` | No | No |
| `fn from(status: StatusMessage) -> PeerSyncInfo` | No | No |
| `fn from(config: &NetworkConfig) -> UPnPConfig` | No | **Yes** |
| `fn from_store_bytes(bytes: &[u8]) -> Result<PersistedDht, StoreError>` | **Yes** | No |

*Table D.1: Functions fuzzed during the network phase of the engagement.*

Fuzzing revealed two different paths to the same underlying bug, which is in `rust-libp2p`'s `serde::Deserialize` implementation for `Multiaddr` (TOB-LIHO-018).

The two `From` trait implementations that were not fuzzed are:

- `fn from(_: tokio::time::Elapsed) -> RPCError`
- `fn from(err: io::Error) -> RPCError`

These were not fuzzed because `serde::Deserialize` is not implemented for either `tokio::time::Elapsed` or `std::io::Error`. It might have been possible to find a workaround—by patching `tokio`, for example. However, we decided that the expected rewards did not outweigh the effort required.