Least Authority
PRIVACY MATTERS

Utility Libraries

Security Audit Report

ChainSafe

Final Report Version: 23 March 2020

Table of Contents

Overview

Background

ChainSafe has requested that Least Authority perform a security audit of their Lodestar utility libraries. Lodestar is an Ethereum 2.0 implementation of the Beacon Chain.

Project Dates

- **February 17 - February 25**: Code review completed *(Completed)*
- **February 28**: Delivery of Initial Audit Report *(Completed)*
- **March 18 - 19:** Verification completed *(Completed)*
- **March 20:** Delivery of Final Audit Report *(Completed)*

Review Team

- Emery Rose Hall, Security Researcher and Engineer
- Jan Winkelmann, Security Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer

Coverage

This report only covers a small subsection of a larger scope that was greatly reduced due to specification changes and outstanding work to be completed.

Target Code and Revision

For this audit, we performed research, investigation, and review of the Lodestar utility libraries followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following components are considered in scope:

- Utility libraries
    - Persistent Merkle Tree: https://github.com/chainsafe/persistent-merkle-tree
    - BLS key derivation and hd key utilities: https://github.com/ChainSafe/bls-hd-key
    - Key management for BLS curves: https://github.com/ChainSafe/bls-keygen
    - BLS key store: github.com/chainsafe/bls-keygen
    - Typescript types for Ethereum 2.0 data structures: https://github.com/ChainSafe/lodestar/tree/master/packages/lodestar-types
    - Utility methods used throughout Lodestar modules: https://github.com/ChainSafe/lodestar/tree/master/packages/lodestar-utils
    - Beacon Chain configuration: https://github.com/ChainSafe/lodestar/tree/master/packages/lodestar-config

However, Ethereum 2.0 BLS Signature Verification, Beacon Chain parameters, Simple Serialize Type Schema, Lodestar, and third party vendor code is considered out of scope.

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

```
bls-hd-key@f772e6673bbb613cf7208648723bd5776596c2fd
```

```
bls-keygen@32b06822b490ee679f7e1334858b8ec5beac5cf1
```

`lodestar/packages/eth2.0-types@6911fc0e4f08678a907afebd778451e7fbee4df4`

`lodestar/packages/eth2.0-utils@8d01a343ada1fac0116b1e6bfeb5bb5746c270d4`

`lodestar/packages/eth2.0-config@8d01a343ada1fac0116b1e6bfeb5bb5746c270d4`

`persistent-merkle-tree@8b5ad7e97e138cdd7770b24c581311cc6bf361de`

For the verification, we examined the Git revision:

`bls-hd-key@767c9989f0dd353249d39e605aababfcbef8413c`

`bls-keygen@32b06822b490ee679f7e1334858b8ec5beac5cf1`

`lodestar@12a2edaecf3910caf7003717ad79cc20a8c2c8fb`

`bcrypto@1b7bcda5cd60db545d6f3425751aee280ce8718a`

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

Supporting Documentation

The following documentation was available to the review team:

- ChainSafe Documentation HackMD: https://hackmd.io/fg3sxYt2RJSKlVSLBb1XYg

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component as well as secure interaction between the network components;
- Data privacy, data leaking, and information integrity;
- Key management implementation: secure private key storage and proper management of encryption and signing keys;
- Storing assets securely;
- Any attack that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;
- General use of external libraries;
- Inappropriate permissions and excess authority;
- Anything else as identified during the initial analysis phase.

Findings

General Comments

We found the packages that were reviewed for this audit to be of exceptional quality. The codebase is logically structured, easy to trace, and comprehensible. This made it a pleasure to evaluate the software for security issues. We also found test coverage to be fair - covering most of the critical code paths. While we did not identify any critical vulnerabilities in the packages we reviewed, we did identify one issue in an upstream dependency that needs attention as well as a few miscellaneous suggestions worth noting.

Specific Issues

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

ISSUE / SUGGESTION	STATUS
Issue A: Key Generation Entropy Source Fallback to `Math.random()`	Resolved
Suggestion 1: Array Creation Syntax	Resolved
Suggestion 2: Dependencies with Known Security Vulnerabilities	Partially Resolved
Suggestion 3: Value Override Edge Case in `objectToCamelCase`	Resolved

Issue A: Key Generation Entropy Source Fallback to `Math.random()`

Location

https://github.com/LeastAuthority/bls-keygen/blob/master/src/index.ts#L12

https://github.com/bcoin-org/bcrypto/blob/master/lib/js/random.js#L21-L22

https://github.com/bcoin-org/bcrypto/blob/master/lib/js/random.js#L157-L167

Synopsis

BLS key generation uses a third party library, bcrypto, to create the key. Bcrypto's `randomBytes()` function unsafely and silently falls back to using `Math.random()` as the entropy source instead of native `crypto` based on the inclusion of a property in the global scope.

Impact

Critical. `Math.random()` is not a cryptographically secure random number generator and its use for anything outside of testing significantly compromises the integrity of generated keys.

Preconditions

An attacker would need to have exploited some type of script injection vulnerability - either XSS, malicious extensions, or rogue dependency.

Feasibility

Low-Medium. The feasibility of such an attack largely depends on the deployment of the code. If running as a regular web application, the attack surface may be much wider than if running as a browser extension with its own global context. In addition, many other factors such as browser, extensions installed, and other factors may radically change the feasibility of this attack for better or worse.

Technical Details

Assuming access to the global scope, an attacker could then nullify the global `crypto.getRandomValues()` function (or if the user is running the application in a browser that does not yet support the WebCrypto API). As a result, the library will automatically and silently fallback to using `Math.random()` if the attacker also satisfies the check `!process.browser && process.env.NODE_TEST === '1'`.

It is true that if an attacker already has access to the global scope they could likely wreak more havoc than merely reducing the integrity of the RNG used for keygen, perhaps overriding `crypto.getRandomValues()` to return a static private key they control. However, an attack such as this is far more likely to be noticed sooner than more covertly reducing the entropy used to generate many users' keys.

Mitigation

We have opened a ticket upstream with bcrypto to remove the use of `Math.random()` from the conditional path in the module and place that testing code in the test suite instead. This would at least prevent the silent compromise of the entropy source by manipulation of the global object.

In the meantime, and perhaps in addition to such an upstream fix, calling `Object.freeze(window.crypto)` to prevent other scripts from attempting to override the `getRandomValues` method. Additionally, having your own check for WebCrypto support is a good path. Using the same check that bcrypto uses:

```
const crypto = global.crypto || global.msCrypto;

const HAS_CRYPTO = crypto && typeof crypto.getRandomValues ===
'function';
```

Performing this check before calling `randomBytes()` and throwing an exception or using SJCL or another crypto library if no crypto support exists is a good patch until something is done upstream.

Remediation

Ultimately, this issue is about being resilient to script injection attacks. This is an area of active research, but great progress has been made with SES (Secure EcmaScript), such as Secure EcmaScript Shim and LavaMoat. Remediation of this issue boils down to further developments and stability in these (and perhaps other) projects, browser-native secure execution sandboxes, etc.

Status

This issue was also reported upstream in bcrypto and was fixed in bcrypto. As a result, `Math.random()`will no longer be a possible fall back path in the case of global scope tampering. Tampering with global variables will now lead to an exception instead of possible silent compromise of the entropy source for keys.

Verification

Resolved.

Suggestions

Suggestion 1: Array Creation Syntax

Location

https://github.com/LeastAuthority/bls-hd-key/blob/master/src/key-derivation.ts#L34

Synopsis

In the function `ikmToLamportSK()`, an array of length 255 is created using the syntax:

```
Array.from(new Array(255), (_, i) => okm.slice(i*32, (i+1)*32));
```

Mitigation

This could be improved for both readability and eliminate the creation of two arrays by using `Array.from()`'s support for array-like objects (those with a length property):

```
    Array.from({ length: 255 }, (_, i) => okm.slice(i*32, (i+1)*32));
```

Status

A pull request has been accepted that removes the double creation of arrays bls-hd-key github.

Verification

Resolved.

Suggestion 2: Dependencies with Known Security Vulnerabilities

Location

https://github.com/LeastAuthority/bls-hd-key/issues/1

https://github.com/LeastAuthority/lodestar/issues/1

Synopsis

A large number of vulnerabilities were found through the use of `yarn audit`, though none were observed to be used outside of development tooling or affect the in scope repositories.

Mitigation

Taking care to keep dependencies up to date when security issues are fixed should be part of the ongoing development process. The risk of known vulnerabilities impacting the codebase can be minimized by upgrading where possible and appropriate. By removing known vulnerabilities, new potential issues will be easier to identify and address.

Status

The ChainSafe team updated several of the core dependencies in the BLS HD-Key repository and has an open pull request for Lodestar that had security issues reported by `yarn audit`. The ChainSafe team has acknowledged that there are more dependencies that need to be updated, some of which may not be currently available to update given that they reside deeper in the dependency tree.

Verification

Partially Resolved.

Suggestion 3: Value Override Edge Case in `objectToCamelCase`

Location

https://github.com/LeastAuthority/lodestar/blob/master/packages/eth2.0-utils/src/misc.ts#L4

Synopsis

The function `objectToCamelCase` accepts an object and converts all property names to camel case format. However, it does not check for name conflicts, so it will blindly override properties if they already exist.

```
    > misc.objectToCamelCase({ some_property: 'foo', someProperty: 'bar' })
    { someProperty: 'foo' }
```

While this is an edge case, and most likely one of low impact, it might be possible to exploit this in tandem with another unknown vulnerability to manipulate data that is processed or displayed to the user.

Mitigation

Check if there is a name conflict before proceeding to set the value. Our suggestion would be to `throw` if there is a conflict. Since this is an unexpected case there is no real way to proceed programmatically with any confidence in which value should be used.

Status

With the introduction of this commit, name conflicts are checked for and `objectToCamelCase` will throw an error if the name already exists on the property.

Verification

Resolved.

Recommendations

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

In addition, we recommend continuing to take an aggressive approach to updating dependencies to resolve outstanding security issues. Pinning to exact versions might be a good approach to retaining more control over when and where upgrades take place. This will help both developers and reviewers to be more proactive about catching new vulnerabilities discovered in dependencies, which is paramount to the security of the code base. However, we acknowledge that there will still be vulnerabilities reported from dependencies deeper in the tree from others' projects and it may not always be feasible to go through each node in the tree to have these updated. This risk should continue to be monitored and mitigated as best practices advise.

As mentioned in the *Coverage* section, this report only covers a small subsection of a larger scope that was greatly reduced due to specification changes and outstanding work to be completed by the ChainSafe team. We strongly recommend a more complete audit of Lodestar, SSZ, and the validator packages before these components are considered for production release.

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.