Least Authority
PRIVACY MATTERS

ethdo

Security Audit Report

Ethereum Foundation

Final Report Version: 17 November 2020

Table of Contents

Overview

Background

Ethereum Foundation has requested a security audit of `ethdo`, a command-line tool for managing common operations on Ethereum 2.0, including creating wallets and accounts, generating data for deposits, and sending exit transactions.

Project Dates

- **September 21 - October 14**: Initial Review *(Completed)*
- **October 20**: Initial Audit Report delivered *(Completed)*
- **November 12 - 16:** Verification Review *(Completed)*
- **November 17:** Final Audit Report delivered *(Completed)*

Review Team

- Ramakrishnan Muthukrishnan, Security Researcher and Engineer
- Jehad Baeth, Security Researcher and Engineer

Coverage

Target Code and Revision

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

The following functions are considered **in-scope** for the review:

Wallet Management
These functions cover wallet creation, import and export, display, etc. The specific in-scope functions are:

- `wallet accounts`: in-scope due to accessing files according to the EIP-2386 and EIP-2680 standards.
- `wallet create`: in-scope due to creating files according to the EIP-2386 and EIP-2680 standards.
- `wallet delete`: in-scope due to accessing files according to the EIP-2680 standard.
- `wallet export`: in-scope due to creating data that should be cryptographically secure and suitable for import.
- `wallet import`: in-scope due to importing data from wallet export.
- `wallet info`: in-scope due to accessing files according to the EIP-2386 and EIP-2680 standards.
- `wallet list`: in-scope due to accessing files according to the EIP-2386 and EIP-2680 standards.

Account Management
These functions cover account creation, import, display, etc. The specific functions are:

- `account create`: in-scope due to creating files according to the EIP-2335 and EIP-2680 standards.
- `account import`: in-scope due to creating files according to the EIP-2335 and EIP-2680 standards.

This audit makes no statements or warranties and is for discussion purposes only.

- `account info`: in-scope due to accessing files according to the EIP-2335 and EIP-2680 standards.
- `account key`: in-scope due to displaying cryptographic data.

Signature Operations
These functions cover creation and verification of signatures. The specific in-scope functions are:

- `signature aggregate`: in-scope due to carrying out cryptographic operations
- `signature sign`: in-scope due to carrying out cryptographic operations
- `signature verify`: in-scope due to carrying out cryptographic operations

Validator Operations
These functions cover creation, monitoring and operations for validators. The specific in-scope functions are:

- `validator depositdata`: in-scope due to carrying out cryptographic and hashing operations that are required to meet the Ethereum 2 phase 0 specification.
- `validator exit`: in-scope due to carrying out cryptographic operations that are required to meet the Ethereum 2 phase 0 specification.

Deposit Operations
These functions cover validation of deposit data. The specific in-scope functions are:

- `deposit verify`: in-scope due to carrying out verification of cryptographic and hashing operations that are required to meet the Ethereum 2 phase 0 specification

Exit Operations
These functions cover validation of exit operations. The specific in-scope functions are:

- `exit verify`: in-scope due to carrying out verification of cryptographic and hashing operations that are required to meet the Ethereum 2 phase 0 specification

Store Type
`ethdo` allows multiple backend stores of encrypted information. These are as follows:

- `filesystem`: in-scope due to being a commonly-used method to access wallets and accounts.
- `remote`: in-scope due to being a commonly-used method to access wallets and accounts.

Dependencies
A considerable amount of the work in `ethdo`, especially the cryptographic work, is carried out by other go modules. The following modules are in-scope for the audit:

- go-eth2-types
- go-eth2-util
- go-eth2-wallet-encryptor-keystorev4
- go-eth2-wallet
- go-eth2-wallet-types
- go-eth2-wallet-hd
- go-eth2-wallet-nd
- go-eth2-wallet-store-filesystem

In addition to the above modules, many EIPs define standards that `ethdo` implements. Adherence to the following standards is in-scope, where applicable:

This audit makes no statements or warranties and is for discussion purposes only.

- EIP-2335
- EIP-2386
- EIP-2680

The following functions are considered **out of scope** for the review:

Account Management
These functions cover account creation, import, display, etc. The specific out of scope functions are:

- `account lock`: Out of scope due to only being used with remote signers.
- `account unlock`: Out of scope only being used with remote signers.

Validator Operations
These functions cover creation, monitoring and operations for validators. The specific out of scope functions are:

- `validator info`: Out of scope due to being purely informational, non-critical, and likely to change with the standardised API.

Node and Chain Information
These functions cover fetching and displaying information from an active beacon node. Due to the lack of a standard API at time of writing, they only support the prysm beacon node. The specific out of scope functions are:

- `block info`: Out of scope due to being purely informational, non-critical, and likely to change with the standardised API.
- `chain info`: Out of scope due to being purely informational, non-critical, and likely to change with the standardised API.
- `chain status`: Out of scope due to being purely informational, non-critical, and likely to change with the standardised API.
- `node status`: Out of scope due to being purely informational, non-critical, and likely to change with the standardised API.

Store Type
`ethdo` allows multiple backend stores of encrypted information. These are as follows:

- s3: Out of scope due to being unused to access wallets and accounts by validator clients.

In addition, third party code that is not stated above is considered out of scope.

Specifically, we examined the Git revisions for our initial review (Git tag v1.5.8):

`0746fa304851f0c63955911270447fe2257b7cd0`

The versions of dependencies for ethdo that are in scope for the audit were picked up from the contents of go.mod file contained in the above repo at that specific revision.

For the verification, we examined the Git revision:

`7391dbe6fba2245023854ac6e9eefb2629534d11`

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:

This audit makes no statements or warranties and is for discussion purposes only.

- README: https://github.com/wealdtech/ethdo/blob/master/README.md
- `ethdo` Docs: https://github.com/wealdtech/ethdo/tree/master/docs
- Ethereum 2.0 Specifications: https://github.com/ethereum/eth2.0-specs

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Random number generation (HD seeds, ND private keys);
- Creation of hierarchical deterministic accounts from a well-known seed are protected against relevant test vectors;
- Key management: secure private key storage and proper management of encryption and signing keys;
- Adherence to the Ethereum 2.0 specification regarding generation of deposit data (each individual component of the deposit);
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Adversarial actions and protection against malicious attacks;
- Vulnerabilities within each component as well as secure interaction with related components;
- Denial of Service (DOS) attacks;
- Protection against malicious attacks and other methods of exploitation;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Review Scope

The `ethdo` implementation adheres to the Ethereum 2.0 Specification for managing common operations on Ethereum 2.0. Ethereum 2.0 presents new concepts and has yet to be tested in production, and as a result, there may be unforeseen challenges and risks following launch. As noted in our audit report for the Ethereum 2.0 Specifications:

"Ethereum 2.0 is one of the first PoS projects planned for production and will likely have the greatest market cap value and the largest number of users at launch. As a result, there have not been many opportunities to study the impacts of design decisions on real world uses of such blockchain implementations, and none at the same scale. Although aspects of the design can be reviewed by comparing them to similar implementations, the collective system may not behave as intended due to the complexity."

The scope of this audit sufficiently covered all security critical components of `ethdo`, including key dependencies from Go Standard Library and internal dependencies written by the `ethdo` development team.

Code Quality + Documentation

The code is very well organized into a number of smaller modules, allowing for easier comprehension and review and limiting the attack surface for potential vulnerabilities. In addition, the `ethdo` implementation

This audit makes no statements or warranties and is for discussion purposes only.

includes comprehensive and substantial code comments, thus adequately explaining the purpose and intended behavior of various functions in the codebase. This proved to be very helpful during the review and minimized confusion or lack of understanding about the intended functionality. And adherence to these development and security best practices, in addition to thorough code reviews, helps to minimize the potential for security vulnerabilities.

Furthermore, while most of the repositories have considerable test coverage, others would significantly benefit from increased testing. In particular, we recommend increasing test coverage for the `ethdo` main repository, `go-eth2-wallet-types`, and `go-eth2-wallet`, in order to assist with early bug detection (Suggestion 3).

We found there to be an absence of high level, project specific design documentation. However, given that most of the codebase implements EIP-2335, EIP-2386 and EIP-2680, adheres to the Ethereum 2.0 Specification, and is very well commented, the lack of high level documentation did not pose an obstacle during our review or thoroughly understanding the intended behavior and functionality. In particular, we found the EIPs to be very helpful during the security review.

Specific Issues

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

ISSUE / SUGGESTION	STATUS
Issue A: Scrypt Parameters are Swapped	Resolved
Issue B: Using Unencrypted Connection to Ethereum Endpoint	Partially Resolved
Suggestion 1: Warn Users Choosing Weak Passphrases	Resolved
Suggestion 2: Explore the Use of Property Based Tests	Unresolved
Suggestion 3: Increase Test Coverage	Resolved

Issue A: Scrypt Parameters are Swapped

Location

https://github.com/LeastAuthority/go-eth2-wallet-encryptor-keystorev4/blob/master/encrypt.go#L32-L33

Synopsis

The `r` and `p` parameters of Scrypt password based key derivation function appear to be swapped. In the above code, `r` is defined as 1 and `p` is 8.

Impact

While `p = 8` is harmless, `r = 1` would result in making the Key Derivation Function (KDF) weak. The parameter `r` is directly proportional to the width of the innermost core hash function's width and also determines the iteration count of the core hash function. The memory usage and CPU time is directly proportional to `r`. As a result, making `r` small increases the Scrypt's weakness.

This audit makes no statements or warranties and is for discussion purposes only.

Preconditions

The attacker has access to the wallet files or gains access to the computer running the wallet.

Technical Details

Making `r` = 1 instead of the recommended value of 8 result in it requiring less memory and less CPU. Attackers can employ more memory and CPU to crack passwords derived from such a Scrypt implementation than on an equivalent function with `r` = 8.

Mitigation

Until this is fixed, existing users should choose longer and non-dictionary word passwords.

Remediation

Make `r = 8` and `p = 1`.

It should be noted that both x/crypto/scrypt docs and RFC 7914 recommend `r = 8`, `p = 1`.

Status

The `ethdo` team has issued a commit which changes the scrypt parameters to `r = 8` and `p = 1`, thus resolving this issue according to the suggested remediation.

Verification

Resolved.

Issue B: Using Unencrypted Connection to Ethereum Endpoint

Location

https://github.com/LeastAuthority/ethdo/blob/0746fa304851f0c63955911270447fe2257b7cd0/cmd/root.go#L369

Synopsis

Passing `WithInsecure()` value to the `grpc.Dial` function returns a `DialOption` value which disables transport security for the client connection.

Impact

This leaves some of the beacon chain communications with the Ethereum network vulnerable to man-in-the-middle (MITM) attacks, thus exposing data to unintended receivers.

Preconditions

In order for the attack to take place, the `ethdo` client and the Ethereum beacon node need to run on separate computers.

Feasibility

Any attacker with network access may be able to sniff the unencrypted data exchange.

Technical Details

Using Golang's `grpc` package with the `DialOption` `WithInsecure()` for the Client results in an unencrypted connection.

This audit makes no statements or warranties and is for discussion purposes only.

Mitigation

Users may configure the `ethdo` client to only connect to the Ethereum beacon node running on the `localhost`.

Remediation

Utilize Go's `gprc` to add a layer of encryption (e.g. Transport Layer Security) compatible with the Ethereum network for the communication between Ethereum node and `ethdo`.

Status

The `ethdo` team has responded to this issue with an update where `ethdo` displays a warning if insecure http connections are made to a non-localhost computer running the Ethereum beacon node. In addition, a command line option called `allow-insecure-connections` has been introduced to allow user consent prior to making an insecure connection.

Ethereum 2.0 will soon support a secure REST-based API for beacon nodes. The `ethdo` team stated that they will pursue this issue with the Ethereum team and will implement a fix once secure REST-based API for beacon nodes usage is available.

Verification

Partially Resolved.

Suggestions

Suggestion 1: Warn Users Choosing Weak Passphrases

Location

https://github.com/LeastAuthority/go-eth2-wallet-encryptor-keystorev4/blob/master/encrypt.go#L44

https://github.com/LeastAuthority/ethdo/blob/master/cmd/passphrases.go

Synopsis

`ethdo` does not accept an empty passphrase, which is a security best practice. However, users are not warned when choosing weak passphrases, such as dictionary words or simple passphrases.

Mitigation

In order to further optimize security, users should be warned when selecting dictionary words or simple passphrases that may be easily brute forced by attackers who have access to the wallet files (e.g. files from stolen computers, disks from old computers, or disks whose past owner stored wallet files but failed to appropriately clear the file system data). Users should be encouraged to use strong, non-dictionary passphrases.

Status

The `ethdo` team implemented changes so that `ethdo` now checks for weak passphrases via the zxcvbn-go library and warns users if such passphrases are chosen.

Verification

Resolved.

This audit makes no statements or warranties and is for discussion purposes only.

Suggestion 2: Explore the Use of Property Based Tests

Location

All repositories.

Synopsis

Property based tests automatically generate tests based on a declarative property of the code. Integration of a property based testing library can help generate many example inputs and check for specific properties. For example, there are `encrypt()` and `decrypt()` functions in `go-eth2-wallet-encryptor-keystorev4` repository. A property of these functions is that an input can go round trip and be encrypted and then decrypted to get back the input. Property based testing libraries can generate a considerable amount of different input and on failure, can shrink the input to the smallest form to help in debugging the problem.

Mitigation

While this is not a security issue, we recommend exploring the use of property based testing, as it increases the likelihood of finding bugs in the handling of edge cases.

Status

The `ethdo` team has responded with the opinion that the `ethdo` codebase is not best suited for property-based tests and that they intend to add property-based tests to the supporting modules (e.g. github.com/wealdtech/go-ecodec).

While updates have not been made at the time of verification, this is not a security critical suggestion.

Verification

Unresolved.

Suggestion 3: Increase Test Coverage

Location

`ethdo`: https://github.com/LeastAuthority/ethdo

`eth2-wallet`: https://github.com/LeastAuthority/go-eth2-wallet

`eth2-wallet-types`: https://github.com/LeastAuthority/go-eth2-wallet-types

Synopsis

Without sufficient test coverage, it is unknown if certain parts of the code will function as intended when a real-world input is exercised. Code coverage is the minimal requirement to ensure that the code works for some inputs. In addition, tests should cover several inputs in order to capture potential edge cases of the code (Suggestion 2). Finally, sufficient code coverage allows future contributors and maintainers of the codebase a degree of confidence that the existing code functions as intended.

The following repositories have sufficient test coverage:

- `go-eth2-util` (~90%)
- `go-eth2-types` (~93%)
- `go-eth2-wallet-encryptor-keystorev4` (~88%)
- `go-eth2-wallet-hd` (~90%)
- `go-eth2-wallet-nd` (~88%)

This audit makes no statements or warranties and is for discussion purposes only.

- `go-eth2-wallet-store-filesystem` (~85%)

The following repositories have insufficient test coverage:

- `ethdo` (main repository has no test coverage)
- `Go-eth2-wallet-types` (no test coverage)
- `go-eth2-wallet` (~10%)

Mitigation

Increase unit test coverage for `ethdo`, `go-eth2-wallet-types and go-eth2-wallet`. Since `ethdo` manages a wallet, increased test coverage results in increased confidence in the tool.

Status

The `ethdo` team has restructured and redesigned the code base for testability.

Verification

Resolved.

Recommendations

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

We recommend the addition of property based tests to the supporting modules, in order to catch edge cases, ensure that the code functions as intended for all inputs, and future contributions and maintainers of the code are confident in the quality of the existing code of which they will expand upon. We also encourage the `ethdo` development team to monitor the progress of the implementation of a secure REST-based API for beacon nodes in Ethereum. Once this is available, we recommend the `ethdo` team update the codebase to reflect this change.

Finally, we commend the `ethdo` development team for strongly considering security, as demonstrated through the quality of the implementation and adherence to the EIPs and specification, in addition to closely following development best practices.

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,

This audit makes no statements or warranties and is for discussion purposes only.

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.

This audit makes no statements or warranties and is for discussion purposes only.

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.

This audit makes no statements or warranties and is for discussion purposes only.