Least Authority
PRIVACY MATTERS

Lisk Project: Protocol Design + Implementation
Security Audit Report

Lisk Foundation

Updated Final Report Version: 18 December 2020

Table of Contents

Overview

Background

Lisk is an open source project focused on blockchain accessibility, running its own blockchain network, Lisk Mainnet, with the LSK native token. The Lisk project further develops the Lisk SDK, which allows developers to easily implement their own blockchains with custom transaction logic.

[Lisk Foundation](#) has requested that Least Authority perform a security audit of the Lisk Project Protocol Design and Implementation.

The following components are considered in scope for the review:

- Lisk Protocol Design
- Lisk SDK 5.0.0: A software development kit for building blockchain applications compatible with the Lisk protocol. It runs in the Node.js runtime, is written in Typescript, utilizes RocksDB for persistent storage, and employs the websocket protocol for P2P communication.
- Lisk Core 3.0.0: The official client for the Lisk Mainnet and Testnet built using the Lisk SDK.

Project Dates

- **October 19 - November 20**: Code review completed *(Completed)*
- **November 25**: Delivery of Initial Audit Report *(Completed)*
- **December 4**: Delivery of Updated Initial Audit Report *(Completed)*
- **December 11-14:** Verification *(Completed)*
- **December 15:** Delivery of Final Audit Report *(Completed)*
- **December 18:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

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

Coverage

Target Code and Revision

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

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

- Lisk SDK 5.0.0: https://github.com/LeastAuthority/lisk-sdk/tree/v5.0.0-alpha.3
- Lisk Core 3.0.0: https://github.com/LeastAuthority/lisk-core/tree/v3.0.0-alpha.4
- Lisk Protocol:
  - Protocol documentation: https://lisk.io/documentation/lisk-protocol/
  - Lisk Improvement Proposals (LIPs): https://github.com/LiskHQ/lips

However, third party vendor code is considered out of scope, along with the following:

- lisk-commander

- lisk-elements subfolders: /lisk-api-client, /lisk-constants, /lisk-client, /lisk-elements, /lisk-passphrase

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

Lisk-sdk: `88b24e03bb28925a036293126dd96ac636218e29`

Lisk-core: `6a1742532104af6f5c010e2ae77d3d982d471751`

For the verification, we examined the Git revision:

Lisk-sdk: `dd3f397d265fe5605dee2f3abf7d218b79234a6d`

Lisk-core: `ada2a7f52950f54b23d68d7cfacb4d5bc9dc5d1c`

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:

- Lisk SDK Documentation: https://lisk.io/documentation/lisk-sdk/
- Lisk Core Documentation: https://lisk.io/documentation/lisk-core/
- Protocol Roadmap: https://lisk.io/roadmap
- Technical documentation of the Lisk SDK: https://docs.google.com/document/d/1MwGGhMs9wCrvfBfwAKMiXGFEJYeMQJhDfMi_PvokPc/edit?usp=sharing

Areas of Concern

Our investigation focused on the following areas:

- General
  - Correctness of the implementation and adherence of the implementation to the specification;
  - Common and case-specific implementation errors;
  - Vulnerabilities within individual components as well as secure interaction between the network components;
  - Inappropriate permissions and excess authority;
  - Data privacy, data leaking, and information integrity;
  - Performance problems or other potential impacts on performance, leading to arbitrarily large bandwidth, computation cost, or halting the network;
  - Attacks that impacts funds, such as the draining or the manipulation of funds;
- Network layer
  - Resistance to Denial of Service (DoS) attacks, eclipse attacks, and other attacks on the network;
  - Malformatted messages that crash peers and unintended peer banning;
- Transaction pool
  - Correct selection of valid transactions for blocks;
  - Robustness against spam or resource depletion attacks;
- Consensus algorithm
  - Implementation satisfies liveness and safety claims;
  - Robustness of standby delegate selection;
- Cryptography

    - Correct use and bindings of cryptographic libraries (`libsodium/node.js` cryptography);
    - Security of forging private keys;
    - Replay attacks;
- Block and transaction processing
    - Block and transaction validation, correct application of blocks and transactions, and correct reverting of transactions and blocks;
- Encoding
    - Correctness and robustness of Protocol Buffers implementation;
- Framework plugins
    - HTTP API: DoS attacks and malformed messages;
    - Forger: Correctness and security of activating/deactivating forging; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Overall, we found the work of the Lisk team highly security-conscious: they had already anticipated several potential attacks and applied mitigations to their codebase before our audit started. We found the scope of the audit to have sufficient coverage to address potential security concerns, especially given that dependencies of the project are limited in number and industry standard.

Approach and Strategy

We used a variety of security tools and techniques during our audit, including fuzz testing of the underlying codec and functions, static analysis tools, security checks against dependencies, and intuitive auditing of the Lisk protocol and implementation. Fuzz testing in dynamic languages is quite different than in compiled languages. However, we were able to find some crashing inputs (Issue A). Static manual analysis revealed no issues, and our review of the dependencies returned no package issues either. Additionally, the Lisk team has pinned their JavaScript dependencies to specific versions, which is a security-conscious practice we generally recommend.

Code Quality & Documentation

The code is well-written and consistently follows industry best practices for TypeScript, such as, using linters and transpilers for building and deployment.

The code is written to be modular and composable, making it logical and organized. The test coverage throughout the SDK and Core repository is sufficient, with several tests for adversarial and security-critical situations. The codebase also has comments explaining the behavior of the different components of the system which aids navigating the codebase for both reviewers and contributors.

The documentation is comprehensive, helpful and accurate. We commend the Lisk team for providing thorough coverage of system design choices and the reasoning of the design decisions. However, there are a few minor typos in the documentation which we recommend correcting (Suggestion 1).

System Design

Peer Handling

The Lisk peer-to-peer networking system uses an unstructured distributed hash table to maintain a record of its peers and their behavior. These peers are sorted and shuffled on a regular basis. Additionally, peers in a node's peer pool are protected from this shuffling based on desirable attributes such as latency, response rate, and netgroup. The latency protections are an interesting choice, as they can only be improved by physically moving a node's location closer to a given target node. From a security perspective, latency measurements can't be faked by the responding node, so shuffling higher latency peers out is a more secure way to select for lower latency nodes without relying on any reporting. In the course of several years, this will likely result in slightly lower average latency for any given node. However, this is based on the assumption that end users are well-distributed.

In delegated proof of stake systems, such as Lisk, eclipse and similar identity-related attacks carry more risk because they tie consensus votes to identities. The Lisk team has designed their peer sorting, routing, and discovery to mitigate against these types of identity attacks, including pre-emptively applying some eclipse attack mitigations that have been utilized in Bitcoin. Additionally, they have designed their delegate voting and selection model to mitigate against this. We recommend monitoring the long-term effects of these mitigation strategies to determine if they perform as intended.

Unexpected Error in the Codec

Empty objects in array type schema properties create a class of errors that are not currently handled in the Lisk-codec module (Issue A). As a result of our discussions during the investigation phase of this report, the Lisk team has decided to remove the cause of this error.

Account Key Derivation and Encryption

PBKDF2, the algorithm used to derive the key used to encrypt the mnemonic, is purely CPU-bound and therefore can be efficiently parallized. This opens an attack vector for searching a low-entropy password space efficiently. A successful attack would allow the attacker having access to the encrypted mnemonic to sign arbitrary transactions on behalf of the target (Issue B). The Lisk Protocol documentation outlines that account keys are derived by extracting the entropy from a BIP39 mnemonic to seed an Ed25519 keypair. This extraction is done using a single application of the SHA-256 algorithm. However, SHA-256 is a hash function and does not provide the properties of an extractor. Although no attacks are known against this use case in general, it is not a recommended cryptographic design (Issue C).

Random Number Generation Scheme

The Lisk protocol uses a deterministic random number generation scheme to seed certain values in the protocol. They have applied mitigations to make last revealer attacks unprofitable, and it's clear that the security in the design of this part of the system has been considered. Nonetheless, we consider this an attack surface for the system and recommend continued caution when introducing any changes or updates to this part of the protocol.

Specific Issues & Suggestions

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

ISSUE / SUGGESTION	STATUS
Issue A: Lisk-codec Unexpected Error	Resolved

Issue B: Lisk-core CLI Mnemonic Encryption	Unresolved
Issue C: Account Key Derivation	Unresolved
Suggestion 1: Correct Typos in Documentation	Resolved
Suggestion 2: Use a Separate Address Key and Signing Key	Unresolved

Issue A: Lisk-codec Unexpected Error

Location

https://github.com/LeastAuthority/lisk-sdk/pull/2

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-codec/src/codec.ts#L104

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-codec/src/collection.ts#L225

Synopsis

Array type schema properties are assigned an empty object by default when no valid element data is present. This causes a class of errors that aren't handled in the Lisk-codec module itself and would therefore have to be handled by the caller.

Impact

As codec is used in multiple places (most notably in transaction and block handling), the impact depends on whether this error is handled, and whether there are any resulting side-effects which might be desirable to an attacker.

Preconditions

An attacker would have to operate a node and manipulate messages (e.g. transactions) to produce this error. An attacker would also likely need to possess or have influence over significant stake in the network to leverage this at higher levels (e.g. block forging), with a potentially greater impact.

Feasibility

This error is possible if an encoding schema with an array property exists.

Mitigation

Disclose this issue to SDK consumers where applicable.

Remediation

We suggest the following:

- Remove the default array element object as mentioned in the discussion on the GitHub PR; and
- Ensure this class of errors is handled in the codec module.

Status

The Lisk team has resolved this issue by removing the initialization of empty objects in the code, which fixes the described error and mitigates the issue as suggested. Additionally, a regression test was added.

Verification

Resolved.

Issue B: Lisk-core CLI Mnemonic Encryption

Location

https://github.com/LeastAuthority/lisk-core/blob/development/src/commands/passphrase/encrypt.ts#L25

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/encrypt.ts#L168

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/encrypt.ts#L115

Synopsis

PBKDF2, the algorithm used to derive the key used to encrypt the mnemonic, is purely CPU-bound and therefore can be efficiently parallized. This opens an attack vector for searching a low-entropy password space efficiently.

Impact

A successful attack would allow the attacker to sign arbitrary transactions on behalf of the target. In the Lisk-core network, actions like token transfers and participation in distributed proof-of-stake could be taken without the target's prior knowledge and without any recourse. In derivative networks, additional actions which utilize account key signatures would also be impacted.

Preconditions

An attacker would need access to the encrypted mnemonic as persisted (cipher-text, salt, iv, etc.), as well as to significant, yet realistic, computational resources.

Feasibility

The preconditions imply some prior successful attack, which makes this issue less likely to be exploited, although still possible. To increase the speed and chance of success, an attacker would likely require substantial hardware acceleration and/or compute resources (e.g. GPUs, FPGAs, or ASICs).

Technical Details

PBKDF2 is being used to derive an AES-256-GCM encryption key from a potentially low-entropy password. That key is then used to encrypt a BIP39 mnemonic. This facilitates file-system persistence of the mnemonic without the need to persist (and secure) an additional key.

The use of password-based key derivation here is a reasonable solution for this case. AES-256-GCM for encryption is a reasonable choice for this use case as well. PBKDF2 however, is purely CPU-bound. This makes it vulnerable to hardware-acceleration-based attacks, especially in the face of a well incentivized and resourceful attacker. Using a memory-hard function prevents this.

Mitigation

We suggest the following mitigation approaches:

- End users could manually secure their encrypted mnemonic with an additional layer of encryption when not in use; or
- The application could estimate the strength of the password and reject ones that are obviously weak; or
- End users could store their encrypted mnemonic on an external device and/or cold storage when not in use.

Remediation

Use the more modern password-based key derivation algorithm `argon2`. Node bindings to the argon2 reference implementation are available under very permissive licenses (MIT, Apache2/CC0).

`Argon2` comes in multiple flavours. Since in this use-case GPU-accelerated attacks are more of a concern than side-channel attacks, the variant `Argon2d` should be used. Section 4 of the Argon2 draft RFC discusses parameter choice and describes a procedure for choosing them optimally.

Status

The Lisk team has responded that they acknowledge that there are better options for the encryption of the mnemonic passphrase, however, they intend to continue using the PBKDF2 algorithm. They note that it is currently used extensively in well-established projects and the risk of the attack is low. As a result, they do not consider the benefit of implementing a new algorithm to outweigh the cost at the present time. Nonetheless, they have indicated that implementing `argon2` has been added to their backlog.

However, they have included in their documentation a reminder to users to always secure their passphrase with a strong password. Although this helps to mitigate the issue, we suggest further action be taken to mitigate the issue in the interim.

Verification

Unresolved.

Issue C: Account Key Derivation

Location

https://github.com/LeastAuthority/lisk-core/blob/development/src/commands/account/create.ts#L30

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/keys.ts#L27

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/keys.ts#L23

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/hash.ts#L26

Synopsis

As described here in the Lisk Protocol documentation, account keys are derived by extracting the entropy from a BIP39 mnemonic to seed an Ed25519 keypair. This extraction is done using a single application of the SHA-256 algorithm. However, SHA-256 is a hash function and does not provide the properties of an extractor. Although no attacks are known against this use case in general, it is not a recommended cryptographic design. Instead, an extractor such as HKDF-Extract should be used to avoid the unnecessary risk for such an attack.

Impact

The derived keys are used to sign network transactions. If compromised, an attacker could sign arbitrary transactions on behalf of the target. In the Lisk-core network, actions like token transfers and participation in delegated-proof-of-stake could be taken without the target's prior knowledge and without any recourse. In derivative networks, additional actions which utilize account key signatures would also be impacted.

Feasibility

No efficient attacks are publicly known.

Technical Details

SHA-256 is a hash function, not an extractor. If you're extracting keys, the best practice is to use an extractor. HKDF is a simple, efficient and provably secure extract-then-expand key derivation function. Many libraries allow running only the extract phase of the KDF, which suffices in this case. The HKDF extract phase takes two inputs: the input keying material IKM and a salt. The purpose of the salt is not to grow the search space, but to randomize the extraction procedure. Therefore, a single 256 bit salt should be chosen at random by the developers and hard-coded in the application.

Remediation

Use HKDF to derive the account key from the mnemonic. Use the mnemonic as input keying material and a random salt that was generated by the developers and hard-coded into the application.

We understand that this is a breaking change for account keys. One migration path could be to operate two accounts in parallel, one of them SHA-based and one HKDF-based, and transfer all assets from the SHA-based account to the HKDF-based account. Then, the SHA-based account would need to be monitored for new incoming transactions, and received assets would need to be forwarded to the HKDF-based account. We recommend generating a new mnemonic for the HKDF-based account, because doing otherwise would constitute a form of key reuse.

Status

The Lisk team responded that they do not intend to implement a remediation for this issue, as they consider the required account migration and added account management effort to be unacceptable at this time. We recommend that the Lisk team reconsider resolving this issue in the future, especially if there is a more convenient opportunity when making other updates to the project.

Verification

Unresolved.

Suggestions

Suggestion 1: Correct Typos in Documentation

Location

https://github.com/LiskHQ/lips/pull/77

https://github.com/LiskHQ/lips/pull/76

https://github.com/LiskHQ/lips/pull/75

https://github.com/LeastAuthority/lisk-sdk/issues/3

https://github.com/LeastAuthority/lisk-sdk/pull/1

https://github.com/LeastAuthority/lisk-core/issues/1

Synopsis

We identified numerous typos in the documentation.

Mitigation

We suggest an additional proofreading of the documentation to fix typos and identify other potential issues.

Status

The Lisk team has updated the documentation correcting the typos as suggested.

Verification

Resolved.

Suggestion 2: Use a Separate Address Key and Signing Key

Location

https://lisk.io/documentation/lisk-protocol/appendix.html#_key_pair_and_address_creation

https://github.com/LeastAuthority/lisk-core/blob/development/src/commands/account/create.ts#L30

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/keys.ts#L27

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/keys.ts#L23

https://github.com/LeastAuthority/lisk-sdk/blob/development/elements/lisk-cryptography/src/hash.ts#L26

Synopsis

As stated in the *Synopsis* of *Issue C*, Lisk account keys are derived from a BIP39 mnemonic using a password-based key derivation scheme referenced in the location. The account key is called such because it is then used to derive the address for the account. The account key is also used to sign transactions in the Lisk network. In the event an account key is compromised, the account holder's only recourse is to create an alternate account and try to submit their transaction before the attacker. Additionally, any identity associated with that account must be abandoned by the user as the attacker now has the ability to act on behalf of the user's account permanently.

Mitigation

Separate the account address from the transaction signing by using two keys instead of one. Between the transaction signer and verifier, only address public key, transaction private and public key, and a cryptographic proof of authorization need to be used. Importantly, the account private key is not required.

Verify the authority of the transaction key via cryptographic proof authorization on behalf of the account key.

This scheme would facilitate the possibility of transaction key revocation. Revoke a compromised transaction key by generating and broadcasting a cryptographic proof of revocation for the transaction key on behalf of the address key. Forgers must then exclude transactions signed by revoked transaction keys. New (or multiple) keys could be generated at any time with their own respective cryptographic proofs or authorization.

Users should be recommended to store their address private key securely elsewhere (i.e. external, cold-storage) and only access it for key management.

Status

The Lisk team has responded that they will carefully consider this suggestion in the context of user accessibility and security benefits. At the time of the verification, the suggestion remains unresolved.

Verification

Unresolved.

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.

Removing the default element object from Lisk-codec will prevent a class of errors that cause a potential attack surface in multiple places in the system. The Lisk team has decided to pursue this remediation route.

We recommend using the more modern, memory-hard password-based key derivation algorithm `Argon2` instead of the CPU-bound PBKDF2 algorithm for the CLI mnemonic encryption in Lisk-core, and an extractor such as HKDF for account key derivation. Additionally, the security of account keys can be increased by using a separate account address key and a transaction signing key instead of a single account key.

We commend the Lisk team for their security-conscious approach and their thorough documentation.

About Least Authority

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

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

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

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

Our Methodology

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

Manual Code Review

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

Vulnerability Analysis

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

Documenting Results

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

Suggested Solutions

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

Responsible Disclosure

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