Lisk Version 4.0 Sapphire Phase

Security Assessment

October 12, 2023

Prepared for:
Oliver Beddows
Lisk Foundation

Prepared by: **Paweł Płatek, Shaun Mirani, Vasco Franco, Bo Henderson**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Lisk Foundation engaged Trail of Bits to review the security of Lisk version 4 (Sapphire phase). Lisk is an ecosystem composed of multiple codebases.

A team of four consultants conducted the review from April 24 to July 7, 2023, for a total of 30 engineer-weeks of effort. Our testing efforts focused on identifying flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. With full access to source code and documentation, we performed static and dynamic testing of several Lisk codebases using automated and manual processes.

Observations and Impact

During the engagement, we found that `lisk-sdk` was well thought out and specified in Lisk Improvement Proposals (LIPs). We did, however, identify some high-severity issues that could compromise the system's confidentiality, integrity, or availability. For example, problems in the sparse Merkle tree implementation could break some cryptographic guarantees of the protocol (TOB-LISK-3, TOB-LISK-4, TOB-LISK-5, TOB-LISK-7). Additionally, issues in how messages are recovered in the interoperability protocol could allow message replaying (TOB-LISK-48). And finally, issues in the node's RPC configuration could allow attackers to expose sensitive key data from the node. We also found the lack of end-to-end tests to be problematic because it made it hard or impossible to properly test the interoperability functionality.

For `lisk-service`, `lisk-desktop`, and `lisk-mobile`—components that interact with `lisk-sdk`—we found the code to be less well specified and of lesser quality (e.g., large amounts of dead code in `lisk-desktop` and the use of JavaScript instead of TypeScript, as explained in TOB-LISK-96).

In these targets, we identified several high-severity issues. In `lisk-service`, we discovered a denial-of-service issue (TOB-LISK-63) and a potential file disclosure issue (TOB-LISK-67). In `lisk-desktop`, we found that Electron best practices were not followed and defense-in-depth mechanisms were lacking (TOB-LISK-53, TOB-LISK-54, TOB-LISK-55, TOB-LISK-56, TOB-LISK-57, TOB-LISK-58, TOB-LISK-61, TOB-LISK-62). We also identified passphrase generation flaws (TOB-LISK-93). In `lisk-mobile`, we discovered problems in how the application validates and displays token amounts coming from the `lisk://` URL protocol (TOB-LISK-90), along with an insecure implementation of biometric authentication (TOB-LISK-76).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Lisk take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Reduce the security responsibility of sidechain developers**. Currently, module developers need to review several security-related tasks, including the following:
  - Validating the parameters received against the command's schema (see TOB-LISK-30)
  - Checking a Cross-Chain Message's (CCM's) status to ensure it is not a bounced message (see TOB-LISK-46)
  - Ensuring the `verify` function for CCMs fails only on transactions that are actually invalid but not on ones that are valid but fail a custom check

  We recommend refactoring the software development kit (SDK) code to remove these responsibilities from developers and place them on the SDK where possible. This will reduce the likelihood of developers creating vulnerable modules that could impact users and Lisk's reputation.

  See the issues linked above for more specific recommendations. See also this bug disclosure in Cosmos where, because modules were expected to validate messages, a missing check led to a critical issue. If refactoring the code to place the responsibility on the SDK is not possible, provide a suite of Semgrep (or similar) static analysis checks to find developer mistakes.
- **Add end-to-end tests to `lisk-sdk`.** The SDK has good unit test coverage but is missing end-to-end tests to analyze properties of a full set of running Lisk nodes. As a result, testing interoperability, for example, is hard. Specifically, we found that one part of the recovery mechanism allowed replay attacks—a high-severity issue—and another part simply did not work (see TOB-LISK-48). This could have been tested effectively only with end-to-end tests. After creating the infrastructure for end-to-end tests, we recommend creating a set of system invariants and tests for each invariant (e.g., a recovered message cannot be recovered again).
- **Ensure LIPs always match the implementation.** Ensure that there is a robust process to keep LIPs up to date with the implementation and vice-versa. We found several discrepancies between the LIPs and the implementation: in some cases, the implementation included code that was not specified in the LIPs (e.g., module endpoints), and sometimes the LIPs included specifications that were not reflected in the implementation (e.g., schema checks). When first creating a LIP specification, it is fine to commit just the LIP; however, after the initial implementation is created, both the LIP and the implementation should always be updated simultaneously. If the specification changes, the implementation should be updated at the same time; if the implementation changes (e.g., a module needs an additional endpoint), the LIP

should be updated simultaneously. When committing implementation changes, link the pull request of the corresponding LIP change, and ensure reviewers confirm their accuracy. Appendix D lists all the discrepancies between the LIPs and the implementation that we found during the audit.

- **Review how schemas are used and validated.** Lisk extensively uses schemas to validate data coming from several origins. This is essential to ensure incoming data is in the correct format (e.g., that a token transfer amount is larger than zero). We identified several issues in schema validation (TOB-LISK-26, TOB-LISK-27, TOB-LISK-28, TOB-LISK-29, and TOB-LISK-36). We recommend creating checks and updating the schema validation code to guarantee at least the following properties:
    - Schemas specified in the code are used.
    - Schemas do not have repeated IDs.
    - Schemas are not copy-pasted.
    - Schemas require every field by default (using an additional keyword to relax the requirement when needed).
- **Review arithmetic operations and the use of numbers in `lisk-sdk` against implicit conversions.** Conversions of a number from `BigInt` or floating point to a constant-sized integer occur when a binary operation is performed on a number. Not accounting for such conversions may easily result in incorrect arithmetic calculations (see TOB-LISK-51).
- **Add fuzzing tests.** We found numerous issues using very simple fuzzing harnesses. We recommend expanding the harnesses provided in this report and implementing additional ones (see appendix E for details). Moreover, fuzz testing should become integrated in Lisk SDK. Lisk SDK should provide a set of methods and utilities for module developers to enable easy integration of fuzz testing. This effort should be accompanied by discovering invariants that must hold for the entire system and every module. Additionally, implement a framework to test the invariants and allow module developers to easily add and test custom invariants.
- **Add Semgrep tests to the CI/CD pipeline.** Our use of Semgrep with a combination of publicly available and custom-written rules uncovered several findings. Incorporating Semgrep into the CI/CD pipeline can help prevent future occurrences of these vulnerabilities. See appendix G for details on the custom Semgrep rules that we created and for ideas for new rules. Maintain a culture of developing new rules for every potentially erroneous pattern and for root cause analysis.
- **Cross-review iOS and Android findings**. If an issue is found in the iOS or Android configuration, a similar issue may also exist in the other configuration. For example,

the iOS configuration includes redundant permissions (TOB-LISK-72), and the Android configuration should be reviewed for similar issues.

- **Cross-review Lisk Desktop and Lisk Mobile findings**. A number of findings reported for `lisk-desktop` may also be relevant for the security of `lisk-mobile`, and vice versa. This is because many functionalities are common to both applications.
- **Clean up `lisk-desktop`'s codebase.** The `lisk-desktop` codebase contains a large amount of dead code (including, for example, vulnerable cryptographic code; see TOB-LISK-96), which makes the repository harder to read and audit . We recommend cleaning up the code from `lisk-desktop` and every other Lisk repository.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	15
Medium	16
Low	32
Informational	30
Undetermined	7

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Authentication	1
Configuration	15
Cryptography	10
Data Exposure	6
Data Validation	57
Denial of Service	4
Error Reporting	1
Patching	2
Undefined Behavior	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Vasco Franco, Consultant
vasco.franco@trailofbits.com

Shaun Mirani, Consultant
shaun.mirani@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 20, 2023	Pre-project kickoff call
April 24, 2023	Code walkthrough meeting
April 28, 2023	Status update meeting #1
May 5, 2023	Status update meeting #2
May 12, 2023	Status update meeting #3
May 22, 2023	Status update meeting #4
May 26, 2023	Status update meeting #5
June 2, 2023	Status update meeting #6
June 12, 2023	Status update meeting #7
June 16, 2023	Status update meeting #8
June 23, 2023	Status update meeting #9
June 30, 2023	Status update meeting #10

July 7, 2023	Delivery of report draft
July 7, 2023	Report readout meeting
October 12, 2023	Delivery of comprehensive report with fix review

Project Goals

The engagement was scoped to provide a security assessment of the Lisk Foundation's Lisk version 4 (Sapphire phase). Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the new interoperability feature secure?
- Can a malicious sidechain steal tokens from the mainchain or other sidechain?
- Can users maliciously mint tokens from terminated sidechains?
- Can a user cause a chain to incorrectly terminate?
- Is the sparse Merkle tree proof verification algorithm correct?
- Is the `Boneh–Lynn–Shacham` (BLS) signatures library used correctly?
- Is the block synchronization mechanism performed in accordance with the specification?
- Are blocks validated correctly?
- Are penalties against misbehaving and malicious nodes applied correctly?
- Are the new modules (e.g., `auth`, `fee`, `token`) prone to attacks?
- Is key material securely stored in the Lisk Desktop and Lisk Mobile applications?
- Can users securely construct transactions in Lisk Desktop and Lisk Mobile if the Lisk Service they connect to is compromised?
- Can users perform denial-of-service attacks against Lisk Service?
- Does Lisk Service expose data that should not be accessible through the API?
- Are Lisk Service security features (e.g., rate limiting) implemented correctly?
- Is a standard Lisk Service deployment secure by default?

Project Targets

The engagement involved a review and testing of the targets listed below.

Lisk SDK

Repository	https://github.com/LiskHQ/lisk-sdk/tree/release/6.0.0
Version	`89e7504ef5eb6183aefe576a93be3d6052e56038`
Type	TypeScript
Platform	Any

Lisk Core

Repository	https://github.com/LiskHQ/lisk-core/releases/tag/v4.0.0-beta.0
Version	`30167c3c340f7758e5788f0b7a98859dcc12dcf0`
Type	TypeScript
Platform	Any

Lisk Improvement Proposals (LIPs)

Repository	https://github.com/LiskHQ/lips
Version	`be768c4fda2df6c46caefe710688314f47d8ceb8`
Type	Documentation
Platform	N/A

Lisk Migrator

Repository	https://github.com/LiskHQ/lisk-migrator
Version	`e53110b1bfcbd691247e8e6ee65d0f17b55483e7`
Type	TypeScript
Platform	Any

Lisk Service

Repository	https://github.com/LiskHQ/lisk-service
Version	`bb4b8e159d6156fccb7c516b71e7c890ec612903`
Type	JavaScript
Platform	Any

Lisk Desktop

Repository	https://github.com/LiskHQ/lisk-desktop
Version	`8238f41dbae0ac55449149db1a19bf0b69cda7d1`
Type	JavaScript
Platform	Any

Lisk Mobile

Repository	https://github.com/LiskHQ/lisk-mobile
Version	`52ac8a151ee4f4fe6f3dbd3f50343ca2052731c9`
Type	JavaScript, React Native
Platform	iOS, Android

Lisk DB

Repository	https://github.com/LiskHQ/lisk-db/releases/tag/v0.3.5
Version	`83234e54bb25ba71f4174da6d34d900dd29c836c`
Type	Rust
Platform	Any

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- Manual review accompanied by static and dynamic testing of LIPs from 0037 to 0071 and their corresponding implementation; the code we audited was fully contained in the following repositories: `lisk-sdk`, `lisk-core`, `lisk-migrator`, and `lisk-db`. All reviewed LIPs are listed below. For clarity, the reviewed LIPs were divided into five groups, based on their functionality:
    - Cryptography
        - LIP-0037: Use message tags and chain identifiers for signatures
        - LIP-0038: Introduce BLS signatures
        - LIP-0039: Introduce sparse Merkle trees
        - LIP-0062: Use pre-hashing for signatures
        - LIP-0064: Disallow non-required properties in Lisk codec
        - LIP-0066: Introduce tree based key derivation and account recovery
        - LIP-0067: Introduce a generic keystore
        - LIP-0068: Define new transaction schema
    - Consensus protocol
        - LIP-0042: Define state transitions of Reward module
        - LIP-0044: Introduce Validators module
        - LIP-0046: Define state and state transitions of Random module
        - LIP-0056: Add weights to Lisk-BFT consensus protocol
        - LIP-0057: Define state and state transitions of PoS module
        - LIP-0058: Define BFT store and block processing logic

        - LIP-0070: Introduce reward sharing mechanism
        - LIP-0071: Introduce dynamic reward module
    - `legacy` module
        - LIP-0050: Introduce Legacy module
        - LIP-0063: Define mainnet configuration and migration for Lisk Core v4
    - State processing
        - LIP-0040: Define state model and state root
        - LIP-0041: Introduce Auth module
        - LIP-0048: Introduce Fee module
        - LIP-0051: Define state and state transitions of Token module
        - LIP-0055: Update block schema and block processing
        - LIP-0060: Update genesis block schema and processing
        - LIP-0065: Introduce events and add events root to block headers
        - LIP-0069: Update Lisk SDK modular blockchain architecture
    - Interoperability
        - LIP-0043: Introduce chain registration mechanism
        - LIP-0045: Introduce Interoperability module
        - LIP-0049: Introduce cross-chain messages
        - LIP-0053: Introduce cross-chain update mechanism
        - LIP-0054: Introduce sidechain recovery mechanism
        - LIP-0059: Introduce unlocking condition for incentivizing certificate generation
        - LIP-0061: Introduce certificate generation mechanism

- In addition to the code related to the LIPs listed above, we audited the following parts of the `lisk-sdk` codebase:
  - Code inside the `elements/lisk-cryptography` folder except legacy methods (e.g., `encryptMessageWithPrivateKey` function)
  - Code inside the `elements/lisk-codec` folder
  - Lisk Validator
  - Lisk RPC functionality
  - Schema definitions
  - Use of various strings encodings
- For the Lisk Desktop (`lisk-desktop`) application, we reviewed the configuration of the Electron application for security best practices, the React application configuration, the creation and storage of mnemonics, sinks that could lead to cross-site scripting (XSS) from the perspective of an external attacker or a malicious sidechain, the `lisk://` URL protocol handler, the WalletConnect integration, the hardware wallet integration, and possible misrepresentations of data in the user interface. Additionally, we ran static analysis tools such as Semgrep and CodeQL and created a fuzzing harness.
- For the Lisk Mobile (`lisk-mobile`) application, the audit included a check for common iOS and Android misconfigurations. We also conducted a review of the app's use of key material, the authentication procedures logic, the custom `lisk://` URL protocol, and the transaction-sending logic from the user interface and external perspectives. Moreover, the Lisk Mobile application was subject to a Data Theorem scan, and all issues reported by the tool were triaged and reported as part of the audit.
- For the Lisk Service (`lisk-service`) application, we looked for patterns vulnerable to regular expression denial-of-service (ReDoS) attacks, audited the security of parameter validation for API calls, and reviewed the MySQL interfacing code, the `blockchain-connector` service, and rate limiting for the JSON-RPC API available over WebSocket. Moreover, we performed static and dynamic testing of the Gateway Service, Export Service, and API parameter validation.

- Manual code review of the above services was accompanied via static and dynamic analyses, including creation of custom static analysis rules (appendix F and appendix G) and fuzzing harnesses for dynamic testing (appendix E).

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Interaction of the sparse Merkle tree (defined in LIP-0039) with databases (in-memory and RocksDB)
- Any component or feature defined in LIPs from 0001 to 0036—specifically, the following directories of `lisk-sdk`, which were not in-scope (the list is not comprehensive):
    - `elements/lisk-api-client`
    - `elements/lisk-client`
    - `elements/lisk-p2p` (peer-to-peer networking)
    - `elements/lisk-transaction-pool`
    - `elements/lisk-transactions`
    - `elements/lisk-tree` (regular Merkle tree)
    - `elements/lisk-passphrase`
    - `elements/lisk-utils`
- Framework plugins and Lisk Commander from `lisk-sdk`, which were not in scope
- The `lisk-mobile` application, which was not comprehensively audited due to time constraints; we focused on the most critical functionalities, which are listed in the previous section. Specifically, the following functionalities need additional review:
    - Integration with WalletConnect
    - Interactions with external services; additional audit of the following areas is recommended: communication and validation of data received from Lisk

Service (in addition to TOB-LISK-83) and communication with other external parties

- Validation of input data provided by a user
- Navigation safety (e.g., phishing opportunities) and general application business logic
- User interface safety (in addition to TOB-LISK-90)
- Networking security (e.g., HTTP client timeouts, handling of HTTP redirects)
- Storage and transition of sensitive data such as mnemonics, passwords, and personally identifiable information (PII)—that is, an additional audit should verify whether such data is stored in-memory redundantly and whether it is stored in the filesystem or other places without relevant security controls
- The attack surface from the perspective of a malicious application running on the same device as Lisk Mobile; a vulnerability of that type was found (TOB-LISK-98), but future investigation for other attack vectors is required.
- Correctness of arithmetic operations, especially integer overflows and implicit conversions

- The custom Lisk application that runs on a ledger device and then interfaces with the Electron application (this application component was out-of-scope)
- Vulnerabilities that were independently discovered by the Lisk team during the audit period, which we have not reported; the following is a non-exhaustive list of such issues:
  - Update the registration order of modules in application
  - Missing generator address and `validatorsHash` check on genesis block
  - `SupportedTokensStore` supports already supported token
  - Failing node synchronization
  - Node generating blocks while synchronizing
  - Sign message from WalletConnect

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix F
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix F
Data Theorem Mobile Secure	A continuous automated security service that finds vulnerabilities and data privacy issues within mobile (iOS and Android) applications	Automatically configured
TruffleHog	A tool for detecting the presence of secrets in version-controlled repositories	Default
Electronegativity	A tool to identify misconfigurations and security anti-patterns in Electron applications	Default
`redos-detector`	A tool for testing regular expressions against ReDoS attacks	Default

Test Results

The results of this focused testing are detailed below.

Lisk-sdk

Property	Tool	Result
Transaction round-trip property: Is the encoding of a decoded transaction equal to the initial transaction?	Jazzer	TOB-LISK-19
Codec round-trip property: Is the encoding of a decoded binary message equal to the binary message?	Jazzer	TOB-LISK-20
Sizes of encoded and decoded objects are valid.	Jazzer	**Passed**
An event class does not have a function caller `error` that adds nonpersistent events (`error_event_is_not_revert` rule).	Semgrep	**Passed**
A stored variable is not read into a variable, modified, and then not stored again (`get_modify_no_set_on_stores` rule).	Semgrep	**Passed**
A store or event is not registered with the incorrect class (`module_registration_of_correct_class` rule).	Semgrep	**Passed**
There are not two stores registered with the same index (`module_stores_same_index` rule).	Semgrep	**Passed**
A schema object does not contain `minItems` and/or `maxItems` on a property that is not of type `array` (`schema_min_max_items_without_array` rule).	Semgrep	TOB-LISK-34
All schema properties have a field number (`schema_property_element_without_field_number` rule).	Semgrep	**Passed**
There are not two properties of the same schema with the same field number (`schema_with_duplicate_field_number` rule).	Semgrep	**Passed**

Schemas require every property (`schema_with_field_not_required` rule).	Semgrep	TOB-LISK-28
Every `verify` function of a module command has a call to `validator.validate` to validate that the received parameters are valid (`verify_without_schema_verify` rule).	Semgrep	TOB-LISK-30
A schema does not require a property that does not exist (`schema_with_required_that_is_not_a_property` rule).	Semgrep	TOB-LISK-26
Schemas use the format attribute only on non-integer types (`schema_int_format_with_integer_type` rule).	Semgrep	TOB-LISK-52

Lisk Desktop

Property	**Tool**	**Result**
The `parseSearchParams` and `stringifySearchParams` functions are round trip.	Jazzer	TOB-LISK-88
No issues are reported by the Electronegativity tool.	Electronegativity	**Failed**

Lisk Mobile

Property	**Tool**	**Result**
No high-severity issues (as determined by the tool) are reported for the Lisk Mobile iOS application.	Data Theorem Mobile Secure	TOB-LISK-72

No high-severity issues (as determined by the tool) are reported for the Lisk Mobile Android application.	Data Theorem Mobile Secure	**Passed**

Lisk Service

Property	**Tool**	**Script**
Schemas do not have hard-coded patterns. Rule `schema_hardcoded_pattern`.	Semgrep	**TOB-LISK-86**
No high-severity issues are reported by the tool.	CodeQL	**Passed**
Patterns are not vulnerable to regular expression denial of service (ReDoS).	redos-detector	**TOB-LISK-63**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Lisk SDK

Category	Summary	Result
Arithmetic	The code handles token amounts using JavaScript's `BigInt` type, which cannot be used or compared with regular numbers. We did not identify division operations that would result in a loss of precision or other type-coercion problems with tokens. The code handles other numeric values using the standard JavaScript numeric types `BigInt` and floating point, sometimes converting them to bit-width limited integers. While most of the arithmetic operations were found to be correct, implicit conversions from `BigInts` and floating-point numbers to bit-width limited integers are not accounted for, which resulted in TOB-LISK-51. We also found that some numeric constants were incorrect (TOB-LISK-12) and that numbers were insufficiently validated in a few code paths (TOB-LISK-24, TOB-LISK-91).	**Moderate**
Auditing	The `lisk-sdk` modules emit on-chain events for errors and operations that occurred, which enables any party to understand the events that happened during a transaction. We did not identify any missing events. The logging coverage (i.e., whether every important action and event in the system emits a log) was not completely reviewed for all system components and thus requires further investigation. Moreover, the security controls for log transition, storage, integrity, retention and rotation, and monitoring	**Further Investigation Required**

	and alerting mechanisms were not audited.	
Authentication / Access Controls	We did not identify any mistakes in how the `auth` module authenticates users and multisignature accounts. We did find configuration issues with the RPC (TOB-LISK-1) that impact access controls.	**Satisfactory**
Complexity Management	The codebase is generally well structured; code responsible for different parts of the application is clearly divided into discrete packages and files.	**Satisfactory**
Configuration	We found flaws in how `lisk-sdk` configures its RPC endpoints (TOB-LISK-1) and stores local files containing sensitive data (TOB-LISK-9). We recommend reviewing the SDK from the perspective of a local attacker operating on the same machine as the `lisk-sdk` application.	**Moderate**
Cryptography and Key Management	Key management (e.g., generation of keys, their storage, backups) was not thoroughly audited and requires further investigation (though TOB-LISK-9 concerns this area). We audited cryptographic primitives but found no major issues.	**Further Investigation Required**
Data Handling	We found several issues related to data handling, including schemas with missing length fields (TOB-LISK-31) or otherwise improper validation (TOB-LISK-27), ambiguous decoding of encoded data such as transactions (TOB-LISK-19, TOB-LISK-20, TOB-LISK-21), and improper duplicate recovery message checks (TOB-LISK-48). Increasing test coverage and implementing fuzzing and static analysis should help increase the maturity in that area. Moreover, changing the design of some data validation routines is recommended (TOB-LISK-30).	**Weak**

Documentation	The Lisk SDK is well documented in LIPs. We found some discrepancies between LIPs and the implementation and also found that some features such as fees could have expanded explanations. We recommend more in-code comments linking to a LIP or a specific pseudocode from a LIP. Ideally, all pseudocode should be easily discoverable in the code.	Satisfactory
Maintenance	We did not review dependencies as part of the audit.	Not Considered
Memory Safety and Error Handling	The Lisk SDK is implemented in a memory-safe language, so memory safety issues were not considered during the audit. The SDK uses exceptions (throw/catch mechanism) to handle errors. While we found no major issues with use of this mechanism, a better error hierarchy could be defined. For example, the Lisk codebase should have a single `root` error type specific to `lisk-sdk`. Then specific modules and classes should inherit from that error type to implement per-module and per-class error types.	Satisfactory
Testing and Verification	Lisk SDK has good coverage of unit tests; however, we found that some functions were missing unit tests (e.g., the `error` method of the `interoperability` module; see TOB-LISK-44), and some mocks were incorrect (e.g. for the `terminateChain` method of the `interoperability` module; see TOB-LISK-40). We also found a lack of end-to-end tests, which made it very hard to test `interoperability` functionality such as recovery messages (TOB-LISK-48). Finally, no fuzz testing or custom rules for Semgrep, CodeQL, or other static analysis engines are used, so issues specific to the codebase could not be identified.	Moderate

Lisk Desktop, Lisk Mobile, Lisk Service

Category	Summary	Result
Arithmetic	Arithmetic operations were not audited and require further investigation.	**Further Investigation Required**
Auditing	The logging coverage (i.e., whether every important action and event in the system emits a log) was not completely reviewed for all system components and thus requires further investigation. The security controls for log transition, storage, integrity, retention and rotation, and monitoring and alerting mechanisms were not audited.	**Further Investigation Required**
Authentication / Access Controls	Lisk Desktop and Lisk Mobile: We discovered several issues in the authentication and access controls. Most importantly, the biometric authentication is incorrectly implemented (TOB-LISK-76), and we found dozens of configuration issues impacting access controls (TOB-LISK-69, TOB-LISK-70, TOB-LISK-75, TOB-LISK-98). A lot of trust is put in the Lisk Service instances that Lisk Mobile receives data from. The instances are implicitly granted a high privilege level, which may break security assumptions that Lisk Mobile and Lisk Desktop users have (TOB-LISK-83). Lisk Service: We did not identify any vulnerabilities related to authentication and authorization.	**Moderate**
Complexity Management	Lisk Desktop: Lisk Desktop contains a large amount of dead code, which makes the code hard to navigate and audit. The dead code also includes vulnerable code for sensitive topics such as transaction signing (TOB-LISK-96). Lisk Mobile: The code is hard to navigate because views are mixed with business logic. However, this issue is mostly unavoidable because of the React-native design. Otherwise, the code is well structured. One suggestion is to move Lisk Service's client code (e.g., the	**Moderate**

	`useNetworkStatusQuery` and `useAuthQuery` methods) to a single directory, which would decrease the amount of code specific to modules (where most of the business logic lies) and so increase code readability. Unused methods and dead code impact code readability. Lisk Service: The codebase is generally well structured; code responsible for different parts of the application is clearly divided into discrete packages and files.	
Configuration	Lisk Desktop: The Lisk Desktop codebase does not take advantage of many Electron features designed to enhance the application's security (TOB-LISK-53, TOB-LISK-54, TOB-LISK-55, TOB-LISK-56, TOB-LISK-57, TOB-LISK-61, TOB-LISK-62). It also exposes its local web server to the local network instead of just on `localhost` (TOB-LISK-58). Lisk Mobile: We found numerous issues with configurations, mostly impacting authentication and access controls. We recommend researching available security-relevant configurations and hardenings and implementing them for both Android and iOS systems. Lisk Service: We identified two misconfigurations in the Docker Compose file that would prevent API rate limiting from being enabled, potentially allowing denial-of-service (DoS) attacks.	**Weak**
Cryptography and Key Management	Lisk Desktop: The cryptographic code in Lisk Desktop has flaws in how mnemonics are generated (TOB-LISK-93) and also contains dead code with vulnerable calls to cryptographic functions. Lisk Mobile: The application uses cryptographic primitives provided by `lisk-sdk`, thereby moving responsibilities to that component. We discovered issues resulting in sensitive data exposure (TOB-LISK-87, TOB-LISK-98), indicating that more investigation in this area is required. General design of key management (mnemonics encrypted with a password, backed up as a file) is solid. However, it puts the burden of storing backups on the users, increasing the possibility of users	**Moderate**

	losing their keys forever. Lisk Service: This area was not considered.	
Data Handling	Lisk Desktop: The codebase fails to validate external data in several instances, including the following: the WalletConnect connection data is not properly validated or displayed to the user (TOB-LISK-80, TOB-LISK-81, TOB-LISK-82, TOB-LISK-84), the `lisk://` URL protocol lacks strict checks (TOB-LISK-61), and several identifiers are not guaranteed to be unique (TOB-LISK-89). Lisk Mobile: Data received from external inputs such as the `lisk://` URL link and Lisk Service is validated to some extent, but as described in TOB-LISK-90 and TOB-LISK-83, the validations seem to be insufficient. Further investigation in this area is required. Lisk Service: Multiple API endpoints fail to sufficiently validate user-supplied data, leading to DoS conditions and a path traversal vulnerability.	**Weak**
Documentation	Lisk Desktop: `lisk-desktop`'s `README.md` file contains documentation on how to install and run the project, along with a description of each of the project's directories. Some functions also have JDocs describing their functionality. We recommend more complete use of JDocs to describe each component. Lisk Mobile: No documentation was reviewed. Lisk Service: The API reference is comprehensive and largely in alignment with the actual requirements and functionality of the code.	**Moderate**
Maintenance	Lisk Desktop: The `lisk-desktop` application contains several outdated dependencies including an old Electron version that has known vulnerabilities. Dependencies of Lisk Mobile and Lisk Service were not audited.	**Further Investigation Required**

Memory Safety and Error Handling	All components are implemented in memory-safe languages, so memory safety issues were not considered during the audit. Lisk Desktop: We found problems where `lisk-desktop` hangs if redirected to an incorrect URL (e.g., `/#/?modal=selectNode`). The `lisk-desktop` application should handle these errors better and redirect the user to the main page when it cannot recover from an error. Lisk Mobile: Further investigation is required. Lisk Service: We discovered one instance of inadequate error handling that would allow an attacker to crash the application.	**Further Investigation Required**
Testing and Verification	Lisk Desktop and Lisk Mobile: Test coverage was not reviewed in detail and requires further investigation. Lisk Service: The current suite of tests was insufficient to identify multiple issues where unexpected or malformed parameter values crashed the application. Furthermore, there was no testing to verify that rate-limiting options are truly enabled when passed to the Gateway service.	**Further Investigation Required**

Maturity of the `lisk-db` codebase is not rated because of the limited scope of our review of that library.

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

Lisk SDK

ID	Title	Type	Severity
1	HTTP and WebSocket API endpoints lack access controls	Access Controls	High
2	Unreachable check for ApplyPenaltyError	Error Reporting	Undetermined
6	Discrepancies between LIPs and the corresponding implementation	Data Validation	Informational
9	Sensitive data stored in world-readable files	Configuration	Low
10	BLS operations involving secret data are not constant time	Cryptography	Low
11	LIP-0066 depends on unfinalized EIP-2333 and EIP-2334	Patching	Informational
12	Invalid argon2id memory parameter	Cryptography	Low
13	Users cannot set argon2id memory limit	Data Validation	Low
14	Log injection via RPC method name	Data Validation	Low
15	RPC request name allows access to object prototype properties	Data Validation	Low
16	XSS in the dashboard plugin	Data Validation	Low

17	Encryption file format proposed in LIP-0067 weakens encryption	Cryptography	**Informational**
18	Pre-hashing enables collisions	Cryptography	**Informational**
19	lisk-codec's decoding method does not validate wire-type data strictly	Data Validation	**Low**
20	lisk-codec's decoding method for varints is not strict	Data Validation	**Low**
21	lisk-codec's decoding method for strings introduces ambiguity due to UTF-8 encoding	Data Validation	**Low**
22	Hash onion computation can exhaust node resources	Denial of Service	**High**
23	setHashOnion RPC method does not validate seed length	Data Validation	**Low**
24	token methods lack checks for negative token amounts	Data Validation	**Medium**
25	token module supports all tokens from mainchain and not just LSK	Data Validation	**Informational**
26	Schemas with required fields of nonexistent properties	Data Validation	**Undetermined**
27	Schemas with repeated IDs	Data Validation	**Undetermined**
28	Schemas do not require fields that should be required	Data Validation	**Low**
29	Lisk Validator allows extra arguments	Data Validation	**Informational**
30	Commands are responsible for validating their parameters	Data Validation	**Medium**

31	Insufficient data validation in validators module	Data Validation	**Informational**
32	Event indexes are incorrectly converted to bytes for use in the sparse Merkle trees	Data Validation	**Medium**
33	Lack of bounds on reward configuration could cause node crashes	Data Validation	**Informational**
34	Mainchain registration schema validates signature length incorrectly	Data Validation	**Informational**
35	Sidechain terminated command uses the wrong chain ID variable	Data Validation	**Low**
36	Hex format validator allows empty and odd-length strings	Data Validation	**Undetermined**
37	CCM fees are always burned	Data Validation	**Low**
38	CCM fees are underspecified and the implementation is not defensive	Data Validation	**Informational**
39	Unspecified order for running interoperability modules	Undefined Behavior	**Informational**
40	The interoperability module's terminateChain method does not work	Undefined Behavior	**Low**
41	Chain ID length not validated in interoperability endpoints	Data Validation	**Informational**
42	Bounced CCM fees are not escrowed	Data Validation	**Low**
43	The send method accepts a status different from OK	Data Validation	**Low**
44	The error method incorrectly checks the status field	Data Validation	**Low**

45	Send method may lead to crash when missing the timestamp parameter	Data Validation	Low
46	The channel terminated command does not validate the CCM status	Data Validation	Low
47	Invalid use of values in the sparse Merkle tree	Data Validation	Low
48	Recovered messages can be replayed	Data Validation	High
49	StateStore handles multiple snapshots dangerously	Data Validation	Informational
50	Invalid base method used in InitializeStateRecoveryCommand	Configuration	Informational
51	Incorrect handling of large integers in regular Merkle tree verification	Data Validation	High
52	Lisk Validator does not validate integer formats when provided as number	Data Validation	Undetermined
91	totalWeight may exceed MAX_UINT64 in mainchain registration command verification	Data Validation	Informational
97	Extraneous feeTokenID option configured for token module	Configuration	Informational
99	LIP-0037 specifies ambiguous requirements for tags	Configuration	Informational
100	BLS library does not properly check secret key	Cryptography	Informational

Lisk DB

ID	Title	Type	Severity
3	Invalid common prefix method	Data Validation	High
4	Panic due to lack of validation for Proof's bitmap length	Data Validation	High
5	Sparse Merkle tree proof's verification algorithm is invalid	Cryptography	High
7	Lack of length validation for leaf keys in sparse Merkle tree proof verification	Data Validation	High
8	Lack of sparse Merkle tree personalized tree-wide constant	Cryptography	Informational

Lisk Desktop

ID	Title	Type	Severity
53	Electron version is outdated and uses vulnerable Chromium version	Patching	Medium
54	Electron renderer lacks sandboxing	Configuration	Low
55	Lack of a CSP in lisk-desktop	Configuration	Low
56	Electron app does not validate URLs on new windows and navigation	Data Validation	Medium
57	IPC exposes overly sensitive functionality	Data Exposure	Low
58	Electron local server is exposed on all interfaces	Configuration	Low

59	Unnecessary use of innerHTML	Data Validation	**Informational**
60	ReDoS in isValidRemote function	Denial of Service	**Informational**
61	Improper handling of the custom lisk:// protocol in lisk-desktop	Data Validation	**Undetermined**
62	Lack of permission checks in the Electron application	Configuration	**Low**
73	Unnecessary XSS risk in htmlStringToReact	Data Validation	**Informational**
80	WalletConnect integration crashes on requiredNamespaces without Lisk	Data Validation	**Informational**
81	WalletConnect integration accepts any namespaces	Data Validation	**Low**
82	Impossible to cancel a WalletConnect approval request without refreshing	Data Validation	**Informational**
83	Desktop and mobile applications do not validate data coming from online services	Data Validation	**High**
84	Users can be tricked into unknowingly authorizing a dapp on a chain ID	Data Validation	**Medium**
88	Missing round-trip property between parseSearchParams and stringifySearchParams	Data Validation	**Undetermined**
89	Several lisk-desktop identifiers are not unique	Data Validation	**Informational**
93	Incorrect entropy in lisk-desktop passphrase generation	Cryptography	**Medium**
94	lisk-desktop attempts to open a nonexistent modal	Data Validation	**Informational**

95	Phishing risk on the deviceDisconnectDialog modal	Data Validation	Low
96	Use of JavaScript instead of TypeScript	Configuration	Low

Lisk Mobile

ID	Title	Type	Severity
68	Mobile iOS application does not use system-managed login input fields	Configuration	Low
69	Mobile iOS application does not exclude keychain items from online backups	Data Exposure	High
70	Mobile iOS application does not disable custom iOS keyboards	Data Exposure	High
71	Mobile application uses invalid KDF algorithm and parameters	Cryptography	Medium
72	Mobile iOS application includes redundant permissions	Configuration	Informational
74	Mobile iOS application disables ATS on iOS devices	Configuration	Low
75	Mobile application does not implement certificate pinning	Cryptography	Low
76	Mobile application biometric authentication is prone to bypasses	Authentication	High
77	Mobile application is susceptible to URI scheme hijacking due to not using Universal Links and App Links	Configuration	Informational

78	Mobile iOS application filesystem encryption is not enabled for locked devices	Data Exposure	**Medium**
79	Mobile Android application permission riding is possible	Access Controls	**Medium**
87	Mobile application caches password in transaction-signing form	Data Exposure	**Medium**
90	Mobile application insufficiently validates and incorrectly displays amount value in transaction transfer	Data Validation	**Medium**
98	Mnemonic recovery passphrase can be copied to clipboard	Data Exposure	**Medium**

Lisk Service

ID	Title	Type	Severity
63	ReDoS in API parameter validation	Denial of Service	**High**
64	HTTP rate-limiting options are not passed to Gateway container	Configuration	**Medium**
65	HTTP rate limiter trusts X-Forwarded-For header from client	Data Validation	**Medium**
66	Unhandled exception when filename for transaction history download is a directory	Denial of Service	**High**
67	Path traversal in transaction history download	Data Validation	**High**
85	Misnamed WebSocket rate-limiting options in Compose file	Configuration	**Medium**

86	Use of hard-coded validation patterns	Data Validation	**Informational**
92	Lack of MySQL LIKE escaping in search parameters	Data Validation	**Informational**

Detailed Findings—Lisk SDK

1. HTTP and WebSocket API endpoints lack access controls	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-LISK-1
Target: `lisk-sdk/framework/src/controller/ws/ws_server.ts`, `lisk-sdk/framework/src/controller/http/http_server.ts`	

Description

A Lisk node can be configured to serve an HTTP- or WebSocket-based JSON-RPC API. The API does not implement any access controls, such as token authentication or `Origin` header allowlisting, that would prevent untrusted web browser resources from communicating with it. As a result, even when the API server is bound only to `localhost`, arbitrary websites can issue API calls and read their responses.

Furthermore, the HTTP server sets an overly permissive cross-origin resource sharing (CORS) policy by sending the `Access-Control-Allow-Origin` header with a wildcard [*], which allows all origins to read server responses (figure 1.1).

```
const headers = {
	'Access-Control-Allow-Origin': '*',
	'Access-Control-Allow-Methods': ALLOWED_METHODS.join(','),
	'Content-Type': 'application/json',
};
```

Figure 1.1: All `Origin` headers are allowed.
(`lisk-sdk/framework/src/controller/http/http_server.ts#51–55`)

Exploit Scenario

An attacker serves a web page containing JavaScript that opens a WebSocket connection to a Lisk node's local JSON-RPC API, executes the `generator_getAllKeys` method, and logs the API's response to the console (figure 1.2).

```
const socket = new WebSocket('ws://localhost:7887/rpc-ws');

socket.addEventListener('open', (event) => {
    socket.send(JSON.stringify({'jsonrpc': '2.0', 'id': 1, 'method':
'generator_getAllKeys'}))
});
```

```
socket.addEventListener('message', (event) => {
    console.log(event.data);
});
```

Figure 1.2: A script on an untrusted web page calls the `generator_getAllKeys` API method via a WebSocket connection.

Alternatively, the attacker can use JavaScript's `XMLHttpRequest` class to make the same API call via an HTTP POST request (figure 1.3).

```
let xhr = new XMLHttpRequest();
xhr.open('POST', 'http://localhost:7887/rpc');
xhr.send(JSON.stringify({'jsonrpc': '2.0', 'id': 1, 'method':
'generator_getAllKeys'}));

xhr.onload = function() {
    console.log(xhr.response);
};
```

Figure 1.3: A script on an untrusted web page calls the `generator_getAllKeys` API method via an HTTP POST request.

A user runs a Lisk node (e.g., using the command `lisk-core start`) with either the HTTP or WebSocket API server listening on `localhost:7887`. On the same machine, the user loads the attacker's web page in their browser. The attacker's script executes, resulting in the server's response, which contains plaintext keypairs (figure 1.4), being logged to the JavaScript console.

```
{"id":1,"jsonrpc":"2.0","result":{"keys":[{"address":"lskzzw334cgd7cew5mb9bbayjmojbo
dgmw9pdym6p","type":"plain","data":{"generatorKey":"1a4e92e2b0c235fff9bcff3b417c5a47
cb5f5240c54f297360a5a516a91bc4fc","generatorPrivateKey":"c77b2f1f79240bcb15a19336beb
d9653886beea724c44a418f0baf26a452dc1f1a4e92e2b0c235fff9bcff3b417c5a47cb5f5240c54f297
360a5a516a91bc4fc","blsKey":"a4f8a7ab03605906f89a6b3e2a6091c53feae76f3a9e48593732fc0
27babbbfd6cf6fa5c98251f8453cb6b57fda99ced","blsPrivateKey":"5787d50e94d4533d46ca9e2d
1b2715bf56a3b4e71bbeed9902b2245a81439721"}},
(...)
```

Figure 1.4: The attacker can read the response for the `generator_getAllKeys` API call.

Recommendations

Short term, add the ability to restrict API access via authentication tokens. Enable authentication by default. Replace the wildcard value [*] in the HTTP endpoint's `Access-Control-Allow-Origin` header (figure 1.1) with a configurable list. Limit the set of HTTP origins permitted to communicate with the WebSocket endpoint to a user-defined list. Alternatively, document risks related to exposing the RPC interface on `localhost`.

Long term, write tests to ensure that untrusted origins cannot communicate with the JSON-RPC API.

2. Unreachable check for ApplyPenaltyError

Severity: **Undetermined**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-LISK-2
Target: `lisk-sdk/framework/src/engine/consensus/consensus.ts`	

Description

In the `Consensus.onBlockReceive` method, a try-catch block around the call to `this._execute(block, peerId)` applies a penalty to a peer if an `ApplyPenaltyError` exception is raised:

```
324    try {
325          const endExecuteMetrics = this._metrics.blockExecution.startTimer();
326          await this._execute(block, peerId);
327          endExecuteMetrics();
328    } catch (error) {
329          if (error instanceof ApplyPenaltyError) {
330                this._logger.warn(
331                      {
332                            err: error as Error,
333                            data,
334                      },
335                      'Received post block broadcast request with invalid
block. Applying a penalty to the peer',
336                );
337                this._network.applyPenaltyOnPeer({
338                      peerId,
339                      penalty: 100,
340                });
341          }
342          throw error;
343    }
```

Figure 2.1: *`lisk-sdk/framework/src/engine/consensus/consensus.ts#324–343`*

However, the highlighted condition is unreachable because none of the statements in the `try` block, nor their callees, throw an instance of `ApplyPenaltyError`. The only references to the `ApplyPenaltyError` symbol in the `lisk-sdk` codebase are in its definition, its import in `consensus.ts`, the above conditional, and the unit tests.

In earlier versions of the codebase, `ApplyPenaltyError` could be thrown by the `_verify` function if the version in a received block's header was invalid, or by the `_validate` function if the transactions contained in a received block were invalid. Block header validation is now performed by the `Chain.validateBlock` method, which does not raise

`ApplyPenaltyError`. Consequently, the `onBlockReceive` method does not detect this condition, and peers may not be appropriately penalized when a block is invalid.

Exploit Scenario

A node receives a block with an invalid block header or transactions. An error condition is raised in one of the functions called by `_execute`, but it is not an instance of `ApplyPenaltyError`, so `onBlockReceive` does not apply a penalty to the misbehaving peer.

Recommendations

Short term, modify all validation routines to ensure that `ApplyPenaltyError` is thrown whenever peer penalization is warranted.

Long term, expand testing to ensure that peers are appropriately penalized for all defined misbehaviors.

6. Discrepancies between LIPs and the corresponding implementation	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-6
Target: `lisk-sdk`, LIPs	

Description
When reviewing LIPs and their implementations, we found several discrepancies between the two. We describe each discrepancy, all of informational severity, in appendix D.

The discrepancies may indicate issues within the Lisk codebase or bugs in the LIPs.

Recommendations
Short term, either fix the code or the LIP specification so that they match.

Long term, create a process where LIP specifications are always created before the implementation is created, including which methods and endpoints exist. If additional functionality must be added, it should be done in parallel with an update to the specification. Only accept implementation pull requests after the LIP has been updated.

9. Sensitive data stored in world-readable files	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-9
Target: `lisk-sdk`	

Description
Lisk creates several local files containing passphrases and private keys with world-readable permissions. These files should be treated as sensitive and readable only by the OS user that created them, similar to how SSH keys are commonly stored.

The `passphrase:create` command creates a passphrase and stores it in a file with world-readable permissions using the `fs.writeJSONSync` function, as shown in figure 9.1.

```
const passphrase = Mnemonic.generateMnemonic(256);

if (output) {
      fs.writeJSONSync(output, { passphrase }, { spaces: ' ' });
```

Figure 9.1: Code that writes a passphrase to a world-readable file (`lisk-sdk/commander/src/bootstrapping/commands/passphrase/create.ts#39–42`)

```
jofra@ubuntu  /tmp/test_passphrase_permissions:
>>> $  hello_client passphrase:create --output ./passphrase.json

jofra@ubuntu  /tmp/test_passphrase_permissions:
>>> $  ll
total 4.0K
-rw-rw-r-- 1 jofra jofra 178 May 12 09:12 passphrase.json
```

Figure 9.2: Image showing the creation of the `passphrase.json` file with world-readable permissions [`-rw-rw-r–`]

The same thing happens in the `keys:create` command and in the `init_generator` logic that sets up the initial configuration for a generator.

Exploit Scenario
A user sets up a Lisk node on a machine shared with other OS users. A malicious user with an account on the same machine reads the user's passphrase and private keys.

Recommendations

Short term, review every use of `fs.writeJSONSync` and other functions that create sensitive files. For each, modify the code to use the `mode` option so that the file permissions allow only the owner to read and write.

Long term, write tests to ensure that every file containing sensitive information has the correct permissions.

10. BLS operations involving secret data are not constant time	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-10
Target: `lisk-sdk/elements/lisk-cryptography/src/bls_lib/lib.ts`, `blst-ts/src/lib.ts`	

Description

A few cryptographic operations dealing with private BLS keys are not constant time. The issue occurs both in the Lisk codebase and in the `blst-ts` dependency.

In the Lisk codebase, the bug is in the `blsSign` function (figure 10.1). The `equals` method compares a private key (`sk` variable) with 32 zero bytes. The method is short-circuiting, returning early on the first nonzero byte found in the key. This behavior discloses partial information about the key if the execution time of the method is measured.

```
export const blsSign = (sk: Buffer, message: Buffer): Buffer => {
	// In case of zero private key, it should return particular output regardless
of message.
	//
elements/lisk-cryptography/test/protocol_specs/bls_specs/sign/zero_private_key.yml
	if (sk.equals(Buffer.alloc(32))) {
		return Buffer.concat([Buffer.from([192]), Buffer.alloc(95)]);
	}

	return Buffer.from(SecretKey.fromBytes(sk).sign(message).toBytes());
};
```

Figure 10.1: The `blsSign` function (`lisk-sdk/elements/lisk-cryptography/src/bls_lib/lib.ts#60–68`)

In the `blst-ts` library, the `SecretKey.fromBytes` function (figure 10.2) has a similar issue. It calls the `isZeroBytes` method (figure 10.3), which has the same behavior as the `equals` method.

```
static fromBytes(skBytes: Uint8Array): SecretKey {
  if (skBytes.length !== SECRET_KEY_LENGTH) {
    throw new ErrorBLST(BLST_ERROR.BLST_INVALID_SIZE);
  }
  if (isZeroBytes(skBytes)) {
    throw new ErrorBLST(BLST_ERROR.ZERO_SECRET_KEY);
  }
  const sk = new SkConstructor();
```

```
  sk.from_bendian(skBytes);
  return new SecretKey(sk);
}
```

Figure 10.2: The `SecretKey.fromBytes` function (`blst-ts/src/lib.ts#63–73`)

```
function isZeroBytes(bytes: Uint8Array): boolean {
  for (let i = 0; i < bytes.length; i++) {
    if (bytes[i] !== 0) {
      return false;
    }
  }
  return true;
}
```

Figure 10.3: Short-circuiting the method called by `SecretKey.fromBytes` (`blst-ts/src/lib.ts#303–319`)

Exploit Scenario

A local attacker measures the execution time of the `blsSign` method. Based on the measurements, he determines the number of leading zero bytes in a private BLS key.

Recommendations

Short term, replace the `equals` method with a constant-time comparison. Work with the `blst-ts` maintainers to fix the issue in the library.

Long term, when writing and reviewing code, make sure that all uses of sensitive data (such as private keys) are constant time.

11. LIP-0066 depends on unfinalized EIP-2333 and EIP-2334	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-LISK-11
Target: LIP-0066, `elements/lisk-cryptography/src/bls.ts`	

Description

LIP-0066 depends on EIP-2333 and EIP-2334. Both EIPs are in `Stagnant` state, which means they are not finalized and may change in the future. For example, they may fix a bug in the `parent_SK_to_lamport_PK` procedure and start using the `index` value as the `info` parameter, instead of the `salt` parameter, in calls to the HKDF.

```
0. salt = I2OSP(index, 4)
1. IKM = I2OSP(parent_SK, 32)
2. lamport_0 = IKM_to_lamport_SK(IKM, salt)
3. not_IKM = flip_bits(IKM)
4. lamport_1 = IKM_to_lamport_SK(not_IKM, salt)
5. lamport_PK = ""
6. for i  in 1, .., 255
        lamport_PK = lamport_PK | SHA256(lamport_0[i])
7. for i  in 1, .., 255
        lamport_PK = lamport_PK | SHA256(lamport_1[i])
8. compressed_lamport_PK = SHA256(lamport_PK)
9. return compressed_lamport_PK
```

Figure 11.1: The `parent_SK_to_lamport_PK` procedure

However, it is unlikely that any breaking changes will be introduced since the current versions have been used to generate all of the Ethereum validator keys to date.

Recommendations

Short term, track changes to EIP-2333 and EIP-2334.

Long term, work with the Ethereum community to finalize the EIPs.

12. Invalid argon2id memory parameter	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-12
Target: `lisk-sdk/elements/lisk-cryptography/src/encrypt.ts`	

Description

LIP-0067 specifies the `argon2id` key derivation function's parameters, as shown in figure 12.1. The parameters are chosen correctly; however, the memory parameter constant used in the code is incorrect.

Recommended Parameters

Argon2id

We recommend using argon2id (instead of PBKDF2) to derive the encryption key, as it is recognised as a more secure key-derivation function (see for example OWASP recommendations). We recommend to follow RFC 9106 for basic parameter choices. Their first recommend options are:

- iterations=1,
- parallelism=4 lanes,
- memory=2048 (2 GiB of RAM),
- 16 bytes salt,
- 32 bytes output.

Figure 12.1: The `argon2id` parameters as specified in LIP-0067

The memory constant is set to a value of `2024` (figure 12.2). The `hash-wasm` library, which implements the `argon2id` method, expects the memory parameter to represent a kilobyte value (figure 12.3).

```
const ARGON2_ITERATIONS = 1;
const ARGON2_PARALLELISM = 4;
const ARGON2_MEMORY = 2024;
```

Figure 12.2: The constant used by Lisk for the memory parameter (`lisk-sdk/elements/lisk-cryptography/src/encrypt.ts#27–29`)

```
async function argon2Internal(options: IArgon2OptionsExtended): Promise<string |
Uint8Array> {
```

```
const { parallelism, iterations, hashLength } = options;
const password = getUInt8Buffer(options.password);
const salt = getUInt8Buffer(options.salt);
const version = 0x13;
const hashType = getHashType(options.hashType);
const { memorySize } = options; // in KB
```

Figure 12.3: The hash-wasm dependency expects the memory parameter to be in kilobytes. (hash-wasm/lib/argon2.ts#114–120)

Lisk effectively uses `argon2id` with 2,024 kilobytes (2,072,576 bytes), instead of 2,048 megabytes (2,147,483,648 bytes, 2,097,152 kilobytes). It is an order of magnitude lower. There is also a typo in the constant (2,024 versus 2,048).

It may not be possible to use 2GiB of memory with the `hash-wasm` library.

Exploit Scenario

An attacker obtains access to a document with an encrypted node's operator keys. The attacker can crack the encryption more easily (with fewer resources) than was desired by the Lisk team.

Recommendations

Short term, replace the `ARGON2_MEMORY` constant with a `2097152` value. If this parameter cannot be used, select another set of parameters from the sets recommended in RFC 9106.

Long term, monitor changes in recommendations for cryptographic parameter choices, as these may evolve as hardware improves.

13. Users cannot set argon2id memory limit

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-13
Target: `lisk-sdk/elements/lisk-cryptography/src/encrypt.ts`	

Description

The `encryptMessageWithPassword` function (with the `encryptAES256GCMWithPassword` alias) takes as an input a map of options, including the `kdfparams.memorySize` option. This option is not used in the function. The `kdfparams.parallelism` option is incorrectly used instead.

```
const iterations =
      kdf === KDF.ARGON2 ? ARGON2_ITERATIONS : options?.kdfparams?.iterations ??
PBKDF2_ITERATIONS;
const parallelism = options?.kdfparams?.parallelism ?? ARGON2_PARALLELISM;
const memorySize = options?.kdfparams?.parallelism ?? ARGON2_MEMORY;
```

Figure 13.1: The `encryptMessageWithPassword` function
(`lisk-sdk/elements/lisk-cryptography/src/encrypt.ts#165–168`)

Exploit Scenario

A user calls the `encryptMessageWithPassword` function with a custom `argon2id`'s options. He believes that the algorithm is using a safe memory value, while in reality it uses 4—the value the user set for parallelism. The user's encrypted data becomes more susceptible to brute-forcing attacks.

Recommendations

Short term, fix the typo in the `encryptMessageWithPassword` function, replacing the `kdfparams?.parallelism` option with `kdfparams?.memorySize` in the correct place.

Long term, add tests for user-adjustable options. Have the code validate options provided by users and, at a minimum, warn them if they choose insecure ones.

14. Log injection via RPC method name

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-14
Target: `lisk-sdk/framework/src/controller/request.ts`	

Description

When the node RPC API processes a request, it tests the user-specified `method` name against a regular expression of valid names, raising an exception if there is no match. The exception message is formatted to include the untrusted `method` name without sanitization (figure 14.1). As a result, the constructed string may contain control characters that enable log injection when the message is printed to the console.

```
public constructor(id: ID, name: string, params?: Record<string, unknown>) {
        assert(
                actionWithModuleNameReg.test(name),
                `Request name "${name}" must be a valid name with module name and action name.`,
        );

        this.id = id;
        [this.namespace, this.name] = name.split('_');
        this.params = params ?? {};
}
```

Figure 14.1: *`lisk-sdk/framework/src/controller/request.ts#33–42`*

Exploit Scenario

An attacker can communicate with the RPC API of a node—for example, by exploiting TOB-LISK-1. The attacker sends the following JSON-RPC message, specifying an invalid `method` name that contains newline characters:

```
{"jsonrpc": "2.0", "id": 1, "method": "\n\nThis entry was created by log injection.\n\n"}
```

Figure 14.2: A JSON-RPC message with control characters (newlines) in the `method` field

When the API receives this message, it identifies the `method` name as invalid and raises the exception shown in figure 14.1. Because the `method` name is prefixed and suffixed with newline characters, the attacker's text (`This entry was created by log injection.`) appears on its own line when printed to the console, giving the node operator the impression that it is a distinct log entry generated by the node.

```
2023-05-11T18:22:02.185Z INFO Work-MBP engine 58560 New web socket client connected
2023-05-11T18:22:02.195Z INFO Work-MBP engine 58560 [status=error err=Request name "

This entry was created by log injection.

" must be a valid name with module name and action name.] Failed to handle WS
request
```

Figure 14.3: The attacker's text appears in the node's logs as its own entry.

Recommendations

Short term, remove control characters (e.g., carriage return [`\r`], line feed [`\n`], etc.) from the `method` name before logging its value.

Long term, consider implementing a safe logging wrapper method that sanitizes all messages by removing control characters.

15. RPC request name allows access to object prototype properties	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-15
Target: `lisk-sdk/framework/src/controller/channels/in_memory_channel.ts`	

Description

To invoke a JSON-RPC request, a node looks up the handler by the name of the request in the `endpointHandlers` object (a dictionary belonging to the `InMemoryChannel` class). If the handler is found, the node calls it, passing as parameters its Bunyan logger object, the parameters of the RPC request, the header of the latest block, the chain ID, and several methods (figure 15.1). The `IPCChannel` class uses a similar construct.

```
if (this.endpointHandlers[request.name] === undefined) {
	throw new Error(
		`The action '${request.name}' on module '${this.namespace}' does not exist.`,
	);
}

const handler = this.endpointHandlers[request.name];
if (!handler) {
	throw new Error('Handler does not exist.');
}

const offchainStore = new StateStore(this._moduleDB);
const stateStore = new PrefixedStateReadWriter(this._db.newReadWriter());
try {
	const result = (await handler({
		logger: this._logger,
		params: request.params ?? {},
		getStore: (moduleID: Buffer, storePrefix: Buffer) =>
			stateStore.getStore(moduleID, storePrefix),
		header: req.context.header,
		getOffchainStore: (moduleID: Buffer, storePrefix: Buffer) =>
			offchainStore.getStore(moduleID, storePrefix),
		getImmutableMethodContext: () => createImmutableMethodContext(stateStore),
		chainID: this._chainID,
	})) as Promise<T>;
```

Figure 15.1: Lookup and invocation of an RPC endpoint handler (`lisk-sdk/framework/src/controller/channels/in_memory_channel.ts#100–124`)

Passing `request.name` directly as the key to `endpointHandlers` (line 106 in figure 15.1) allows access to the object prototype for the dictionary if `request.name` is the name of a valid JavaScript object property (e.g., `constructor`, `toString`, `valueOf`). As a result, these properties are unintentionally exposed to invocation through the RPC API.

Exploit Scenario

An attacker can communicate with the RPC API of a node—for example, by exploiting TOB-LISK-1. The attacker sends the following JSON-RPC message, specifying `app_constructor` as the RPC `method` in order to invoke the constructor of the `endpointHandlers` object:

```
{ "jsonrpc": "2.0", "id": 1, "method": "app_constructor", "params": {}}
```

Figure 15.2: JSON-RPC request that calls the constructor of `endpointHandlers`

Upon receiving the request, the API looks up and calls the constructor of `endpointHandlers`, passing the parameters shown on lines 115–123 of figure 15.1. A new object is created with these parameters, converted to JSON, and returned to the attacker. Included in the parameters is information that is not intended to be exposed over the API, such as the hostname of the node and the PID of the Lisk process (figure 15.3).


```
{"id":1,"jsonrpc":"2.0","result":{"logger":{"_events":{},"_eventsCount":0,"_level":2
0,"streams":[{"type":"raw","level":20,"stream":{"_trace":false},"raw":true,"closeOnE
xit":false}],"serializers":{},"src":false,"fields":{"name":"application","hostname":
"Work-MBP","pid":63344},"haveNonRawStreams":false},"params":{},"header":{"version":2
,"timestamp":1682608040,"height":7088,"previousBlockID":{"type":"Buffer","data":[44,
126,43,163,242,221,39,13,120,117,174,175,140,155,201,94,236,105,242,189,123,181,182,
53,186,243,21,231,241,97,43,232]},
...
```


Figure 15.3: The API invokes the constructor and returns a new object in JSON form.

Recommendations

Short term, in the `BaseChannel` class, modify the type of `endpointHandlers` to be a `Map`, and use the `get` method for lookups. Alternatively, in the `InMemoryChannel` and `IPCChannel` classes, rewrite the lookup of `request.name` in `endpointHandlers` to use the `hasOwnProperty` method (figure 15.4). Using `hasOwnProperty` prevents the attack described above because it returns `true` only if an object has the specified property set as its own property, rather than having inherited it from `Object.prototype`.

```
if (this.endpointHandlers.hasOwnProperty(request.name)) {
    handler = endpointHandlers[request.name];
} else {
    throw new Error('Handler does not exist.');
}
```

Figure 15.4: Using `hasOwnProperty` to prevent access to the object prototype

16. XSS in the dashboard plugin

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-16
Target: `lisk-sdk/framework-plugins/lisk-framework-dashboard-plugin`	

Description

The dashboard plugin renders data in the text area for call-endpoint arguments in a way that allows the execution of JavaScript. This text area highlights the inserted JSON using the `json-format-highlight` library, which does not properly escape HTML and, in combination with the use of `dangerouslySetInnerHTML`, causes the injection.

```
<code dangerouslySetInnerHTML={{ __html: showHighlightJSON(props.value) }} />
```

Figure 16.1: Code that allows the injection of JavaScript (`lisk-sdk/framework-plugins/lisk-framework-dashboard-plugin/src/ui/components/input/TextAreaInput.tsx#84`)

Figure 16.2 shows the exploit triggering when the payload `{"asd": <img src=x onerror=alert('hello')>asd</img> }` is pasted into the vulnerable text area.

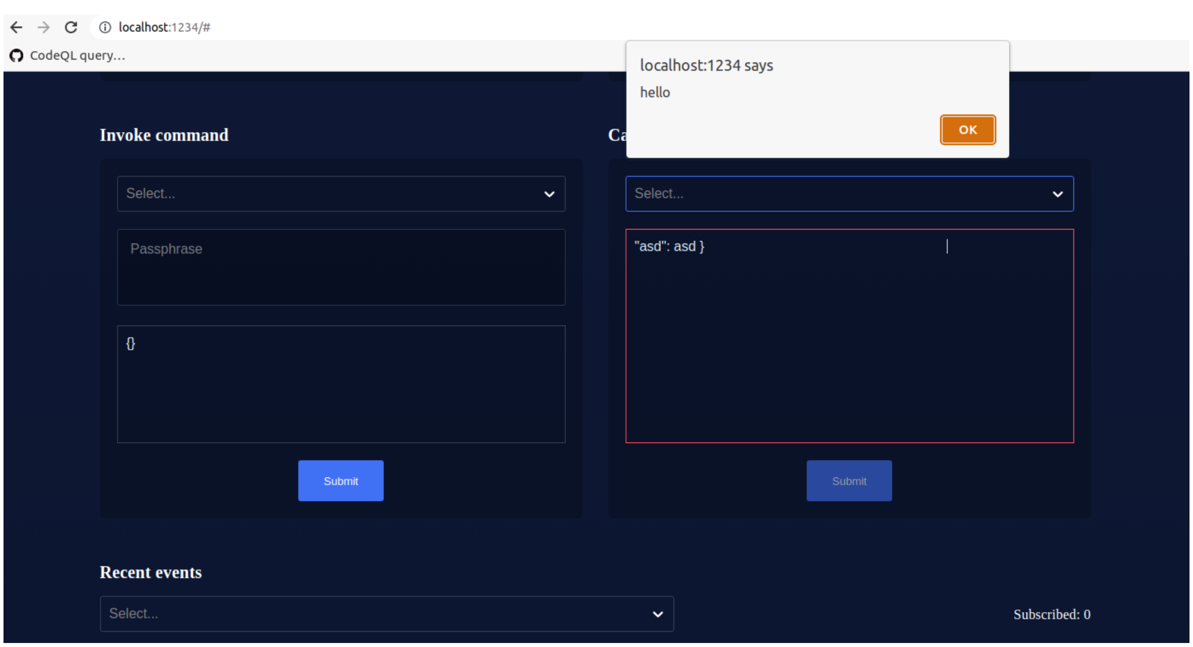

Figure 16.2: Triggering cross-site scripting (XSS)

The same problem exists in the `RecentEventWidget` element, which is responsible for showing the recent events in the dashboard. However, in this element, the attacker controls only part of the JSON contents, which means exploitation may not be possible. Further analysis must be done to determine exploitability.

Code inside the `lisk-sdk/framework-plugins` folder was not in scope for the audit; this issue was found with a static analysis tool.

Exploit Scenario

A user wants to call the `token_getBalance` endpoint. He knows the endpoint call needs some parameters, so he uses Google to find and copy a template of those parameters (expecting to modify the address field to his own address). He pastes the contents into the vulnerable text area, executing the attacker's payload, which steals the contents of the passphrase text area that contains the user's passphrase.

Alternatively, an attacker can create a transaction that emits an event with his payload (e.g., the `hello` module tutorial). When viewing the recent events in the dashboard, the user is exploited.

Recommendations

Short term, write code to sanitize the input value before passing it to the `showHighlightJSON` function, instead of relying solely on `showHighlightJSON` to sanitize the HTML.

Long term, on every use of `dangerouslySetInnerHTML`, document in the code why it is fine to do so in that particular case.

17. Encryption file format proposed in LIP-0067 weakens encryption

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-17
Target: LIP-0067, `lisk-sdk/elements/lisk-cryptography/src/encrypt.ts`	

Description

The Encryption File Format section of LIP-0067 defines a `mac` property, which is computed as a hash of the concatenation of half of an encryption key and encrypted data. Because only half of the key is used, it is significantly easier to brute-force the key (given encrypted data) compared to brute-forcing the entire key. This issue weakens the security of the encryption to 128 bits.

```
return {
	ciphertext: encrypted.toString('hex'),
	mac: crypto.createHash('sha256').update(key.slice(16,
32)).update(encrypted).digest('hex'),
	kdf,
	kdfparams: {
		parallelism,
		iterations,
		memorySize,
		salt: salt.toString('hex'),
	},
	cipher: Cipher.AES256GCM,
	cipherparams: {
		iv: iv.toString('hex'),
		tag: tag.toString('hex'),
	},
	version: ENCRYPTION_VERSION,
};
```

Figure 17.1: Computation of the `mac` property
(`lisk-sdk/elements/lisk-cryptography/src/encrypt.ts#186–202`)

The finding has been set only to informational severity because the 128 bits of security may be considered good enough, especially when compared to the security of a user-provided password protected with a key derivation function (KDF).

Exploit Scenario

An attacker steals a document in the encryption file format from LIP-0067. The document contains an encrypted node's operator keys. The attacker cracks the 16 bytes of the encryption key using the document's `mac` property and cracks the remaining 16 bytes using the encrypted data.

Recommendations

Short term, have the code include the entire key as part of the hash for the `mac` field, or remove the `mac` field altogether.

Long term, use private cryptographic keys for a minimal set of tasks, ideally using a single key for only one functionality (e.g., signing, encryption). For every use of a key outside of its main functionality, review and document potential security consequences.

18. Pre-hashing enables collisions

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-18
Target: LIP-0062, `lisk-sdk/elements/lisk-cryptography/src/ed.ts`	

Description

Pre-hashing data for signing—as specified in LIP-0062—introduces the risk of data collision and the risk of domain separation being negated. A message signed with pre-hashing may have the same signature as the hash of the message signed without pre-hashing. The issue is even more prevalent because double pre-hashing is used for some signatures. Possible confusion vectors include the following:

- If a user signs a raw, 32-byte-long message, the signature can be used as the signature of a pre-hashed or double pre-hashed message. Signing raw, 32-byte-long messages is currently not supported in the Lisk SDK. However, it could have been possible before LIP-0062 was created, or it could become possible in the future. Additionally, a user may misuse their private key to sign bytes without pre-hashing.
- If a user signs a tagged, 32-byte-long message, the signature can be used in the context of a double pre-hashed signature. The tagged message would collide with the internal hash function of the `digestMessage` function.

```
export const signDataWithPrivateKey = (
	tag: string,
	chainID: Buffer,
	data: Buffer,
	privateKey: Buffer,
): Buffer => signDetached(hash(tagMessage(tag, chainID, data)), privateKey);

export const signData = signDataWithPrivateKey;
```

Figure 18.1: Signing with a single pre-hashing
(`lisk-sdk/elements/lisk-cryptography/src/ed.ts#162–169`)

```
export const digestMessage = (message: string): Buffer => {
	const msgBytes = Buffer.from(message, 'utf8');
	const msgLenBytes = encodeVarInt(message.length);
	const dataBytes = Buffer.concat([
		SIGNED_MESSAGE_PREFIX_LENGTH,
		SIGNED_MESSAGE_PREFIX_BYTES,
		msgLenBytes,
```

```
            msgBytes,
        ]);

        return hash(hash(dataBytes));
};
```

Figure 18.2: The `digestMessage` function used by the `signMessageWithPrivateKey` function to prepend the string `Lisk Signed Message:` before double pre-hashing (`lisk-sdk/elements/lisk-cryptography/src/ed.ts#39–50`)

Moreover, the Ed25519 signature algorithm itself comes with two versions: pre-hashed and pure. But because the algorithm uses a different hash function (SHA-512), it is not impactful. Changing Lisk's default hash function to SHA-512 would enable more confusion vectors.

Exploit Scenario

An attacker performs a birthday attack to find a collision between a hash of a message starting with the `Lisk Signed Message:` string and data 21 bytes long (32 bytes long when prepended with a tag and chain ID). The attacker tricks a user into signing the data and broadcasting the signature. The attacker then uses the signature as it would be for the `Lisk Signed Message:` string.

Recommendations

Short term, document the risks related to pre-hashing and double pre-hashing. Have the application warn users against signing data that is not pre-hashed (especially if it is 32 bytes long). Consider replacing the double hash in the `digestMessage` function with a single hash, and ensure that the `SIGNED_MESSAGE_PREFIX_BYTES` constant is a unique tag when compared with tags used in the `signData` function.

Long term, review cryptographic protocols implemented in Lisk against collision and domain-separation attacks resulting from divergent uses of hash functions.

19. lisk-codec's decoding method does not validate wire-type data strictly	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-19
Target: `lisk-sdk/elements/lisk-codec/src/keys.ts`	

Description

LIP-0064 specifies the following round-trip property to hold for every valid message:

```
encode(decode( binaryMsg )) == binaryMsg
```

The `encode` and `decode` methods are implemented in the `lisk-codec` package. The property does not hold because of the lack of strict validation of wire-type information during decoding.

The `readKey` method—shown in figure 19.1—decodes the `fieldNumber` and `wireType` fields from an encoded object's key. There are two wire types: zero for strings, bytes, objects, and arrays; and two for other data types. The `readObject` method, which uses the `readKey` method, does not check whether the decoded wire type matches the expected one. For example, an encoding schema that expects a string value for a given key incorrectly accepts a wire type of zero.

```
const WIRE_TYPE_TWO = 2; // string, bytes, object, array
const WIRE_TYPE_ZERO = 0; // uint32, uint64, sint32, sint64, boolean

export const readKey = (value: number): [number, number] => {
	const wireType = value & 7;
	if (wireType === WIRE_TYPE_TWO || wireType === WIRE_TYPE_ZERO) {
		const fieldNumber = value >>> 3;

		return [fieldNumber, wireType];
	}

	throw new Error('Value yields unsupported wireType');
};
```

Figure 19.1: The `readKey` method
(`lisk-sdk/elements/lisk-codec/src/keys.ts#17–29`)

A unit test in figure 19.2 demonstrates the issue.

```
const binaryMsg =
hexToBuffer("0800101120000000100b700fc005700cbff011ae5180020002800300003a003a00");
const tx = Transaction.fromBytes(binaryMsg); // decode
const binaryMsg2 =  tx.getBytes(); // encode
expect(Buffer.compare(binaryMsg, binaryMsg2)).toBe(0);
```

Figure 19.2: A unit test demonstrating violation of the round-trip property

Exploit Scenario

An attacker creates two transactions with different byte representations that decode to the same transaction. He broadcasts the first transaction to some nodes and the second to other components of the Lisk system. The transaction is included in a block. Lisk system users are confused because the on-chain transaction has different encoding than the transaction they see in the other components.

Recommendations

Short term, add stricter validation for wire types in `lisk-codec`'s decoding methods.

Long term, use fuzz testing to test the round-trip property.

20. lisk-codec's decoding method for varints is not strict

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-20
Target: `lisk-sdk/elements/lisk-codec/src/varint.ts`	

Description

The `lisk-codec` package's methods for decoding varints—`readUInt32`, `readUInt64`, `readSInt32`, and `readSInt64`—do not strictly require a varint to be in the shortest (canonical) form. This behavior breaks the round-trip property specified in LIP-0064 (see TOB-LISK-19).

For example, a canonical encoding of an unsigned 32-bit integer with a value of 75 is `\x4b`, but encodings with appended zero bytes, such as `\xcb\x00`, are also accepted.

```
export const readUInt32 = (buffer: Buffer, offset: number): [number, number] => {
	let result = 0;
	let index = offset;
	for (let shift = 0; shift < 32; shift += 7) {
		if (index >= buffer.length) {
			throw new Error('Invalid buffer length');
		}
		const bit = buffer[index];
		index += 1;
		if (index === offset + 5 && bit > 0x0f) {
			throw new Error('Value out of range of uint32');
		}
		result = (result | ((bit & rest) << shift)) >>> 0;
		if ((bit & msg) === 0) {
			return [result, index - offset];
		}
	}
	throw new Error('Terminating bit not found');
};
```

Figure 20.1: One of the methods implementing varint decoding (`lisk-sdk/elements/lisk-codec/src/varint.ts#62–80`)

Exploit Scenario

An attacker creates two transactions with different byte representations that decode to the same transaction. He broadcasts the first transaction to some nodes and the second to other components of the Lisk system. The transaction is included in a block. Lisk system users are confused because the on-chain transaction has different encoding than the transaction they see in the other components.

Recommendations

Short term, add stricter validations for varints in `lisk-codec`'s decoding methods. Consult ADR 027, which is linked in the References section, for proposed validations.

Long term, use fuzz testing to test the round-trip property.

References

- ADR 027: Deterministic Protobuf Serialization, Cosmos SDK

21. lisk-codec's decoding method for strings introduces ambiguity due to UTF-8 encoding	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-21
Target: `lisk-sdk/elements/lisk-codec/src/string.ts`	

Description

The `lisk-codec` package supports UTF-8 encoded strings. Due to how UTF-8 encoding works, the round-trip property specified in LIP-0064 (see TOB-LISK-19) is violated.

```
export const readString = (buffer: Buffer, offset: number): [string, number] => {
	const [value, size] = readBytes(buffer, offset);
	return [value.toString('utf8'), size];
};
```

Figure 21.1: `lisk-codec`'s method for decoding strings
(`lisk-sdk/elements/lisk-codec/src/string.ts#22–25`)

Consider the UTF-8 decoding example in figure 21.2. The input data is a single `\xff` byte, which constitutes an invalid character in UTF-8 encoding. When interpreted as a UTF-8 string, the `\xff` byte will be replaced with the � replacement character. Then when encoded back to `bytes`, it will become `\xef\xbf\xbd`.

```
it('utf8 test', () => {
	const input = Buffer.from([0x01, 0xff]);
	const [s, ] = readString(input, 0);
	const output = writeString(s);
	expect(input).toBe(output);
});
```

Figure 21.2: UTF-8 decoding ambiguity example

To observe ambiguity in UTF-8 encoding, consider a unit test in figure 21.3. Two input strings with a value of a¨, but with different byte representations, are converted to distinct buffers. The possibility of encoding the same character in various ways is an inherent UTF-8 feature. The feature does not break the round-trip property of LIP-0064 but may cause confusion for humans.

```
it('utf8 test 2', () => {
	const input = "ä";
	const input2 = "ä";
```

```
        const output = writeString(input);
        const output2 = writeString(input2);
        expect(Buffer.compare(output, output2)).toBe(0);
});
```

Figure 21.3: UTF-8 encoding ambiguity example

Exploit Scenario

An attacker creates two transactions with different byte representations that decode to the same transaction. He broadcasts the first transaction to some nodes and the second to other components of the Lisk system. The transaction is included in a block. Lisk system users are confused because the on-chain transaction has different encoding than the transaction they see in the other components.

Recommendations

Short term, conduct research on how to correctly handle UTF-8 strings. At a minimum, the ambiguities shown in this finding's Description section should be addressed, but there are more such ambiguities. For example, lengths of strings in TypeScript may be unexpected. This behavior could impact validations such as maximum allowed lengths.

Long term, use fuzz testing to test the round-trip property.

References

- Unicode Technical Report #36: UNICODE SECURITY CONSIDERATIONS
- UTF-8 Everywhere
- Characters that byte, GoSecure

22. Hash onion computation can exhaust node resources	
Severity: **High**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-LISK-22
Target: `lisk-sdk/framework/src/modules/random/endpoint.ts`, `lisk-sdk/commander/src/bootstrapping/commands/hash-onion.ts`, `lisk-sdk/elements/lisk-cryptography/src/utils.ts`	

Description

The `hashOnion` function implements the hash onion computation outlined in LIP-0022. The function calculates the number of SHA-256 hashes specified in the `count` parameter and stores a subset of them in memory, based on the value of the `distance` parameter.

```
for (let i = 1; i <= count; i += 1) {
	const nextHash = hash(previousHash).slice(0, HASH_SIZE);
	if (i % distance === 0) {
		hashes.push(nextHash);
	}
	previousHash = nextHash;
}
```

Figure 22.1: The `hashOnion` function computes the specified number of hashes and stores some of them in memory.
(`lisk-sdk/elements/lisk-cryptography/src/utils.ts#L197–203`)

While the default hash count, and the one defined in the specification, is one million, `hashOnion` accepts arbitrarily large values for this parameter. The function is reachable from the `setHashOnion` RPC method of the `random` module and from the `HashOnionCommand` class of Lisk Commander, neither of which impose an upper limit on the `count` value provided by the user.

Because the hash onion computation is a blocking operation, a sufficiently large value for the `count` parameter can cause the loop in figure 22.1 to iterate for an inordinate amount of time, resulting in a denial of service to all other node functionality. Furthermore, because a subset of the computed hashes is stored in memory, a large value for `count` can cause the node to exhaust its available memory and crash.

Exploit Scenario

An attacker can communicate with the RPC API of a node—for example, by exploiting TOB-LISK-1. The attacker invokes the `random_setHashOnion` RPC method, providing a count value of 1.7976931348623157e+308 (equivalent to JavaScript's Number.MAX_VALUE constant):

```
{"jsonrpc": "2.0", "id": 1, "method": "random_setHashOnion", "params":{"address":
"lskkgfk84gngotqjsqfujkctshvkwzs8z8w93gmja", "count": 1.7976931348623157e+308}}
```

Figure 22.2: The random_setHashOnion RPC call with a large value for count

The node begins to compute the specified number of hashes, blocking execution of the rest of the process while doing so. Eventually, the node exhausts its available memory and crashes (figure 22.3).

```
<--- Last few GCs --->

[9851:0x7fb5f2300000]   154156 ms: Scavenge 4041.5 (4118.3) -> 4038.6 (4121.3) MB,
17.3 / 4.3 ms  (average mu = 0.451, current mu = 0.291) allocation failure
[9851:0x7fb5f2300000]   154208 ms: Scavenge 4044.2 (4121.3) -> 4041.0 (4138.5) MB,
15.1 / 4.1 ms  (average mu = 0.451, current mu = 0.291) allocation failure
[9851:0x7fb5f2300000]   156003 ms: Mark-sweep 4054.1 (4138.5) -> 4046.2 (4146.8) MB,
1706.9 / 14.5 ms  (average mu = 0.378, current mu = 0.208) allocation failure
scavenge might not succeed


<--- JS stacktrace --->

FATAL ERROR: Reached heap limit Allocation failed - JavaScript heap out of memory
 1: 0x103c2f665 node::Abort() (.cold.1)
[/Users/shaun/.nvm/versions/node/v16.20.0/bin/node]
...
```

Figure 22.3: The node eventually crashes after running out of memory.

Recommendations

Short term, define and enforce an upper bound on the count value that a user can provide to the random_setOnionHash RPC method and the hash-onion command. Define an upper bound on the amount of memory the computation should require, and reject values of the count and distance parameters that exceed this bound.

Long term, consider reimplementing hashOnion to execute in a non-blocking manner.

23. setHashOnion RPC method does not validate seed length	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-23
Target: `lisk-sdk/framework/src/modules/random/endpoint.ts`, `lisk-sdk/elements/lisk-cryptography/src/utils.ts`	

Description

According to LIP-0022, the seed for the hash onion computation should be a 16-byte value. However, the `setHashOnion` RPC method that allows callers to execute this computation does not ensure that the provided seed is 16 bytes long. The validation schema does not specify a required length for the `seed` property (figure 23.1), and the `hashOnion` function that performs the computation does not check the `seed` property's length.

As a result, a user can provide a valid hex string of any length for the seed, which the method will accept and use for the computation. Shorter (and therefore lower-entropy) seed values than allowed by the specification may weaken the security properties of the hash onion computation.

```
export const hashOnionSchema = {
	$id: 'lisk/random/setSeedRequestSchema',
	type: 'object',
	title: 'Random setSeed request',
	required: ['address'],
	properties: {
		address: {
			type: 'string',
			format: 'lisk32',
		},
		seed: {
			type: 'string',
			format: 'hex',
		},
		count: {
			type: 'integer',
			minimum: 1,
		},
		distance: {
```

```
                    type: 'integer',
                    minimum: 1,
                },
            },
        };
```

Figure 23.1: There is no length requirement for the `seed` property of the `hashOnionSchema` schema. (`lisk-sdk/framework/src/modules/random/schemas.ts#27–50`)

Exploit Scenario

A user inadvertently calls the `setHashOnion` RPC method with a hex-encoded seed value less than 32 characters (16 decoded bytes) long. This lower-entropy value violates the assumptions made in LIP-0022 and weakens the security properties of the hash onion computation.

Recommendations

Short term, add a requirement to `hashOnionSchema` for the `seed` property to be exactly `SEED_LENGTH` * 2 characters long (to account for hex encoding). Have the code validate the length of the `seed` property in the `hashOnion` function.

Long term, write tests to verify that `setHashOnion` and `hashOnion` reject invalid seed lengths.

24. token methods lack checks for negative token amounts	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-24
Target: `lisk-sdk/framework/src/modules/token`	

Description

All `token` module methods (i.e., functions that other modules can call) that receive an amount parameter fail to check whether this amount is negative, even though the `BigInt` type can represent negative numbers. These methods include `mint`, `burn`, `transfer`, `lock`, and `unlock`.

This behavior allows several invariants to be broken, including the following:

- The `transfer` method may transfer funds from the receiver to the sender. This can be triggered with a call to `transfer(sender, receiver, -10)`.
- The `burn` function may increase the total supply and increase a user's available balance. This can be triggered with a call to `burn(-10)`.
- The `mint` function may reduce the total supply and reduce a user's available balance. This can be triggered with a call to `mint(-10)`.
- The `lock` and `unlock` functions may be used as their counterparts. This can be triggered with a call to `lock(-10)` or `unlock(-10)`.

These methods are not reachable directly through a `transfer` command because the schema validates that the amount parameter is a `uint64`. However, because of the lack of data validation, vulnerable modules that interact with the `token` module are more likely to contain exploitable bugs. In other locations, the code has defense-in-depth logic to protect against vulnerable modules (e.g., locked amounts are stored by module so that module A cannot unlock funds locked by module B because of a bug), but this is not the case for a negative amount of tokens.

Exploit Scenario

A custom module that interacts with the `token` module calls the `transfer` method from the malicious user's account to another account. Because of a bug in the custom module, the malicious user can fully control the transfer value, including negative values. They trigger this bug and steal tokens from other users by transferring negative amounts of tokens.

Recommendations

Short term, in all the functions described above, implement checks to ensure that an amount is always positive. Write unit tests to ensure that using negative amounts results in an error.

Long term, review every module's methods to ensure inputs are always validated. A module has several trust boundaries: the external boundary, where users can interact with the module through transaction commands; the RPC boundary, where a smaller subset of users can interact with the module, usually to retrieve information; and the module-to-module boundary, where modules interact with each other through method calls. In the latter, modules are not expected to be malicious; however, we recommend designing the method interface to prevent a vulnerable module from breaking the main module's invariants. We recommend validating all the methods' input data, with a schema or otherwise, so that bad inputs are detected sooner, preventing exploits.

25. token module supports all tokens from mainchain and not just LSK	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-25
Target: `lisk-sdk/framework/src/modules/token`	

Description

LIP-0051 states that “all chains must support their native tokens and all chains must support the LSK token.” However, to check whether a token is supported by default, the code uses the `_isMainchainOrNative` function, which accepts any token of the mainchain, not just LSK (whose `tokenID` is `0`).

```
private _isMainchainOrNative(tokenID: Buffer): boolean {
	const [chainID] = splitTokenID(tokenID);
	if (chainID.equals(this._ownChainID)) {
		return true;
	}

	return chainID[0] === this._ownChainID[0] &&
getMainchainID(chainID).equals(chainID);
}
```

Figure 25.1: The `_isMainchainOrNative` function is not strict enough.
(`lisk-sdk/framework/src/modules/token/stores/supported_tokens.ts#192–199`)

This finding is set to informational because LSK is the only token on the mainchain at the moment; however, to align with LIP-0051, the code should be more precise.

Recommendations

Short term, support only the LSK token by having the code check that the `tokenID` associated with the mainchain ID is zero, LSK’s `tokenID`.

26. Schemas with required fields of nonexistent properties	
Severity: **Undetermined**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-26
Target: `lisk-sdk`, Lisk's schemas	

Description

The Lisk codebase contains schemas where the `required` field includes properties that do not exist in the associated `properties` object.

Figure 26.1 shows an example where the `totalSupply` field is required but not included in the `properties` field. Conversely, the `amount` property is missing from the `required` field.

```
export const getEscrowedAmountsResponseSchema = {
	$id: '/token/endpoint/getEscrowedAmountsResponse',
	type: 'object',
	properties: {
		escrowedAmounts: {
			type: 'array',
			items: {
				type: 'object',
				required: ['escrowChainID', 'totalSupply', 'tokenID'],
				properties: {
					escrowChainID: {
						type: 'string',
						format: 'hex',
					},
					tokenID: {
						type: 'string',
						format: 'hex',
					},
					amount: {
						type: 'string',
						format: 'uint64',
					},
				},
			},
		},
	},
};
```

Figure 26.1: `lisk-sdk/framework/src/modules/token/schemas.ts#L585-L611`

The problem also exists in the `getPendingUnlocksResponseSchema` schema.

We wrote tests to ensure that the `validator` would throw an error in these cases, as shown in figure 26.2. This confirmed that the `validator` correctly refuses to use schemas with an invalid `required` field, so the schema described above is never compiled, which can be a symptom of other problems.

```
expect(() =>
    validator.compile(getEscrowedAmountsResponseSchema),
).toThrow(
    'strict mode: required property \"totalSupply\" is not defined at
\"/token/endpoint/getEscrowedAmountsResponse#/properties/escrowedAmounts/items\"
(strictRequired)',
);

expect(() =>
    validator.compile(getPendingUnlocksResponseSchema),
).toThrow(
    'strict mode: required property \"amount\" is not defined at
\"modules/pos/endpoint/getPendingUnlocksResponse#\" (strictRequired)',
);
```

Figure 26.2: Test showing that the validator correctly refuses to use the malformed schemas

Another problem impacting several schema objects is the `required` field being misplaced. For arrays, the `required` field should be placed inside the `items` object. We found several schemas with misplaced `required` fields. Furthermore, the misplacement does not lead to a compilation error (e.g., in case of undefined properties).

```
it('misplaced required property in array types', () => {
    const testSchema = {
        type: 'object',
        required: ['testArray'],
        properties: {
            testArray: {
                type: 'array',
                fieldNumber: 1,
                required: ['myProp'],
                items: {
                    type: 'object',
                    properties: {
                        myProp: {
                            dataType: 'string',
                            fieldNumber: 1,
                        },
                    },
                },
            },
        },
    };

    expect(() =>
        validator.validate(
```

```
            testSchema,
            {
                testArray: [
                    { "test": 123 },
                ],
            },
        ),
    ).toThrow();
});
```

Figure 26.3: Test that fails because the `testArray` property is not checked for the required myProp property

The code paths with misplaced `required` fields are as follows:

- `lisk-sdk/framework-plugins/lisk-framework-forger-plugin/src/schemas.ts#L31-L48`
- `lisk-sdk/framework/src/modules/random/schemas.ts#L166-L191`

We found these issues with the help of the following Semgrep rule:

```
rules:
  - id: schema_with_required_that_is_not_a_property
    message: The required property $PROP_A does not exist
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            {
                ...,
                required: [..., '$PROP_A', ...],
                ...,
            }
        - pattern-not-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
                required: [..., '$PROP_A', ...],
                ...,
            }
        - focus-metavariable:
            - $PROP_A
```

Figure 26.4: Semgrep rule that detects required properties that are not in the `properties` field

Exploit Scenario

A command expects its schema to validate the presence of a given parameter. Because the `required` field includes the wrong property name, an attacker can send a malformed message that breaks some command assumptions, causing it to perform unintended actions.

Recommendations

Short term, remove keys in the `required` field that are not in the `properties` field, and always place the `required` field in the `items` property of an array. Research why the schemas that would throw an error if compiled are not being used and determine whether they should be used or removed.

Long term, integrate the Semgrep rule from figure 26.4 in the CI/CD pipeline to detect instances of this issue before they are committed. Ideally, guarantee that all schemas are compiled so that these errors do not occur during execution.

27. Schemas with repeated IDs	
Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-27
Target: `lisk-sdk`, Lisk's schemas	

Description

A schema's `$id` field is used to cache the schema's compilation and then retrieve it on a future validation. So if two different schemas have the same `$id`, the code may validate the first one and then, when validating the second one, use the (incorrectly) cached version of the first one.

We found completely different schemas with the same `$id`, likely caused by a copy-paste error:

- `$id: '/block/header/3'` in `lisk-sdk/elements/lisk-chain/src/schema.ts#L102-L104` and `lisk-sdk/elements/lisk-chain/src/schema.ts#L112-L114`
- `$id: '/jsonRPCRequestSchema'` in `lisk-sdk/framework/src/controller/jsonrpc/utils.ts#L27-L28` and `lisk-sdk/framework/src/controller/jsonrpc/utils.ts#L49-L50`
- `$id: '/modules/random/endpoint/isSeedRevealRequest'` in `lisk-sdk/framework/src/modules/random/schemas.ts#L308-L309` and `lisk-sdk/framework/src/modules/random/schemas.ts#L324-L325`
- `$id: '/token/endpoint/getBalance'` in `lisk-sdk/framework/src/modules/token/schemas.ts#L494-L495` and `lisk-sdk/framework/src/modules/token/schemas.ts#L448-L449`

We also found the following schemas, which are identical and have the same `$id`:

- `$id: '/auth/account'` in `lisk-sdk/framework/src/modules/auth/schemas.ts#L22-L23` and `lisk-sdk/framework/src/modules/auth/stores/auth_account.ts#L25-L26`
- `$id: '/block/event'` in `lisk-sdk/framework/src/abi/schema.ts#L30-L31` and `lisk-sdk/elements/lisk-chain/src/schema.ts#L187-L188`

- `$id: '/block/event/standard'` in `lisk-sdk/elements/lisk-chain/src/schema.ts#L226-L227` and `lisk-sdk/elements/lisk-api-client/src/codec.ts#L41-L42`

- `$id: '/commander/plainGeneratorKeys'` in `isk-sdk/commander/src/bootstrapping/commands/keys/encrypt.ts#L37-L38` and `lisk-sdk/commander/src/bootstrapping/commands/keys/create.ts#L24-L25`

- `$id: '/modules/random/seedReveal'` in `lisk-sdk/framework/src/modules/random/stores/validator_reveals.ts#L29-L30` and `lisk-sdk/framework/src/modules/random/schemas.ts#L196-L197`

- `$id: '/token/store/escrow'` in `lisk-sdk/framework/src/modules/token/schemas.ts#L109-L110` and `lisk-sdk/framework/src/modules/token/stores/escrow.ts#L23-L24`

- `$id: '/token/store/supply'` in `lisk-sdk/framework/src/modules/token/schemas.ts#L77-L78` and `lisk-sdk/framework/src/modules/token/stores/supply.ts#L21-L22`

These issues show that these schemas are repeated in multiple places instead of imported from a single point. This can cause problems where one version is updated (e.g., with a new length check) while the other is not. This is exemplified in the following locations where both versions of the schema have the same `$id` and are similar but slightly diverged:

- `$id: '/node/forger/usedHashOnion'` in `lisk-sdk/framework/src/modules/random/stores/used_hash_onions.ts#L30-L32` and `lisk-sdk/framework/src/modules/random/schemas.ts#L276-L278`

- `$id: '/token/store/user'` in `lisk-sdk/framework/src/modules/token/stores/user.ts#L27-L28`, `lisk-sdk/examples/interop/pos-mainchain-fast/config/scripts/recover_lsk_plugin.ts#L68-L69`, and `lisk-sdk/framework/src/modules/token/schemas.ts#L52-L53`

Exploit Scenario

The code validates the schema for one command that accepts negative amounts. A second command mistakenly uses the same schema ID but supports only unsigned integers for amounts. The first command's schema is cached, and when the parameters of the second command are validated, a negative value is accepted, leading to a loss of funds.

Recommendations

Short term, remove all instances of schemas with repeated IDs. If the schemas are identical, place the schema in one file and import from the other locations. If the schemas are similar but have slight differences, choose the correct one, place it in one file, and import from the other locations. If the schemas are different, replace the `$id` of one of them.

Long term, write a test that gathers all schema `$id`s and throws an error if it finds two or more identical ones.

28. Schemas do not require fields that should be required	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-28
Target: `lisk-sdk`, Lisk's schemas	

Description

Schema validation does not enforce the existence of each property of the object being validated. However, in most cases, a developer intends for each property to be required; for this reason, most property names (but not all) are included in the `required` field.

We found several instances of properties that should be required but are not. For example, the `mandatory` array in figure 28.1 can have objects with only the `publicKey` field, only the `signature` field, or no field at all. This should not be allowed.

```
export const sortMultisignatureGroupRequestSchema = {
	$id: '/auth/command/sortMultisig',
	required: ['mandatory', 'optional'],
	type: 'object',
	properties: {
		mandatory: {
			type: 'array',
			items: {
				type: 'object',
				properties: {
					publicKey: {
						type: 'string',
						minLength: ED25519_PUBLIC_KEY_LENGTH * 2,
						maxLength: ED25519_PUBLIC_KEY_LENGTH * 2,
						fieldNumber: 1,
					},
					signature: {
						type: 'string',
						minLength: ED25519_SIGNATURE_LENGTH * 2,
						maxLength: ED25519_SIGNATURE_LENGTH * 2,
						fieldNumber: 2,
					},
				},
			},
			<REDATED>
```

Figure 28.1: `lisk-sdk/framework/src/modules/auth/schemas.ts#L107-L157`

It is hard to exhaustively list all instances of this issue because some properties are meant to be optional. The Semgrep rule to find all schema properties that are not required is shown in figure 28.2.

```
rules:
  - id: schema_with_field_not_required
    message: Field $PROP_A is not required
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
            }
        - pattern-not-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
                required: [..., '$PROP_A', ...],
                ...,
            }
        - focus-metavariable:
            - $PROP_A
```

Figure 28.2: Semgrep rule that detects properties that are not required

Exploit Scenario

A command expects a parameter to exist. An attacker sends a transaction without the expected parameter, which breaks the command's assumptions and potentially leads to exploitable bugs.

Recommendations

Short term, run the Semgrep rule shown in figure 28.2, identify the true positives, and correct the schemas.

Long term, consider defaulting every property to be required. `Ajv` does not support this by default, so every schema would need to pass through a preprocessing step where the required property is injected for every property of every object. Additionally, an optional keyword could be introduced for the rare case where the schema needs to have optional parameters.

29. Lisk Validator allows extra arguments	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-29
Target: `lisk-sdk/elements/lisk-validator`	

Description

The `lisk-validator` application's `validate` function does not prevent extra arguments in the object from being validated. Figure 29.1 shows a test that fails because the `validate` call containing the extra `redundant` argument does not throw an error.

```
it('test if extra arguments are possible', () => {
    const validSchema = {
        type: 'object',
        properties: {
            myProp: {
                dataType: 'string',
                fieldNumber: 1,
            },
        },
        required: ['myProp'],
    };

    expect(() =>
        validator.validate(
            schema,
            { myProp: 'test', redundant: 'test' },
        ),
    ).toThrow();
});
```

Figure 29.1: Test that fails because validating the problematic object does not throw an error

The issue is set only to informational because the `lisk-codec` package does not allow constructing (decoding) objects with redundant fields.

Recommendations

Short term, add the `additionalProperties: false` attribute to the schema. This will prevent additional parameters from being accepted.

Long term, in the same preprocessing step recommended in TOB-LISK-28, inject the `additionalProperties: false` attribute in every schema.

30. Commands are responsible for validating their parameters	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-30
Target: `lisk-sdk`, module commands	

Description

Module commands are responsible for validating their parameters (typically in the `verify` function). It is the responsibility of module developers to always remember to validate the schema, but this lack of automation will inevitably lead to bugs. For example, we followed the `hello module tutorial`, which forgot this schema validation step, and found that `hello` messages can exceed their minimum and maximum lengths (the `transaction:create` command validates the schema, but an attacker can encode the transaction without it).

To avoid this issue, the schema could be verified in the state machine when creating the verify-command context. Currently the `createCommandVerifyContext` function only decodes the parameters according to the schema (figure 30.1), but it could also validate them.

```
params: (paramsSchema
      ? codec.decode(paramsSchema, this._transaction.params)
      : undefined) as T,
```

Figure 30.1: Code that decodes the parameters according to their schema (`lisk-sdk/framework/src/state_machine/transaction_context.ts#L120-L122`)

The same issue exists when validating cross-chain messages (CCMs); validating the schema is the responsibility of CCM developers. Instead, the schema could be validated where it is currently only decoded in the `apply` and `applyRecovery` functions.

To find instances where Lisk developers forgot to add this validation step in their own module commands, we wrote the Semgrep rule shown in figure 30.2.

```
rules:
  - id: verify_without_schema_verify
    message: Found a verify function without a call to validator.validate(...)
    languages: [typescript]
    severity: WARNING
    patterns:
        - patterns:
            - pattern-inside: >
```

```
                verify(...) {
                  ...
                }
            - pattern-not-inside: >
                verify(...) {
                  ...
                  validator.validate(...);
                  ...
                }
```

Figure 30.2: Semgrep rule that detects `verify` functions without a call to `validator.validate`

We found the following issues:

- The `BaseCCRegistrationCommand` module command calls `codec.decode` on the `CCUpdateParams` and `CCMRegistrationParams` schemas. This means that the lengths of several properties that have minimum and maximum lengths will not be validated.

- The `SidechainCCSidechainTerminatedCommand` command does not validate the `CCMSidechainTerminatedParams` schema. This means that the length of the `chainID` and `stateRoot` properties will not be checked.

- The `TerminateSidechainForLivenessCommand` command does not validate the `TerminateSidechainForLivenessParams` schema. This means that the length of the `chainID` property will not be checked.

We also observed that the `RegisterKeysCommand` of the `legacy` module validates its arguments in the `execute` function instead of the `verify` function, which wastes computing power in the case of malformed parameters.

Exploit Scenario

A developer forgets to add a schema validation check in their module command. An attacker circumvents limits such as the `maxLength` limit. This breaks assumptions of the command and leads to an exploitable bug.

Recommendations

Short term, implement the validation check on the state machine, possibly in the `createCommandVerifyContext` function. Do the same for CCMs in the `apply` and `applyRecovery` functions to remove this responsibility from developers and guarantee that schemas are always validated.

Long term, find other instances where developers are responsible for checks that could be refactored to the SDK. If any are found, refactor them.

References

- Ethan Buchman, “Cosmos-SDK & IBC Vulnerability Retrospective: Security Advisories Dragonberry and Elderflower (October 2022),” Issue #2: Elderflower, Cosmos Hub Forum, December 16, 2022.

31. Insufficient data validation in validators module	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-LISK-31
Target: `lisk-sdk/framework/src/modules/validators/method.ts`, `lisk-sdk/framework/src/modules/validators/endpoint.ts`	

Description

Several methods of the `validators` module do not validate all parameters according to the requirements in LIP-0044:

- The `registerValidatorKeys` method of the `ValidatorsMethod` class does not ensure that the length of the `proofOfPossession` parameter is equal to the `BLS_POP_LENGTH` constant.
- The `setValidatorBLSKey` method of the `ValidatorsMethod` class does not ensure that the length of the `proofOfPossession` parameter is equal to `BLS_POP_LENGTH`.
- The `validateBLSKey` method of the `ValidatorsEndpoint` class does not ensure that the length of the `proofOfPossession` parameter is equal to `BLS_POP_LENGTH`, as the `validateBLSKeyRequestSchema` schema does not specify a required length for this field.
- The `validateBLSKey` method of the `ValidatorsEndpoint` class does not ensure that the length of the `blsKey` parameter is equal to `BLS_PUBLIC_KEY_LENGTH`, as `validateBLSKeyRequestSchema` does not specify a required length for this field.

Recommendations

Short term, have the code validate the length of all data fields by adding the appropriate `length` property to the schema and calling the `liskValidator.validate` method on the received objects. Where no schema exists for a structure or field, have the code manually validate the parameters, as is already done for certain parameters in the `ValidatorsMethod` class.

32. Event indexes are incorrectly converted to bytes for use in sparse Merkle trees	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-32
Target: `lisk-sdk/elements/lisk-chain/src/event.ts`	

Description

The `keyPair` method of the `Event` class—used to construct and verify sparse Merkle tree roots of on-chain events—incorrectly converts event indexes to bytes. If the `keyPair` method is called on a large number of events but less than the `MAX_EVENTS_PER_BLOCK` constant specified in LIP-0065, an error is thrown.

The issue is that the bitwise left-shift operation shown in the highlighted part of the code in figure 32.1 produces a negative `indexBit` for indexes (`this._index` variable) greater than or equal to 2^{29}. The `writeUIntBE` method throws an error for negative numbers. The currently allowed maximum number of events (`MAX_EVENTS_PER_BLOCK`) is 2^{30}-1.

```
public keyPair(): { key: Buffer; value: Buffer }[] {
	const result = [];
	const value = utils.hash(this.getBytes());
	for (let i = 0; i < this._topics.length; i += 1) {
		// eslint-disable-next-line no-bitwise
		const indexBit = (this._index << EVENT_TOPIC_INDEX_LENGTH_BITS) + i;
		const indexBytes = Buffer.alloc(EVENT_TOTAL_INDEX_LENGTH_BYTES);
		indexBytes.writeUIntBE(indexBit, 0, EVENT_TOTAL_INDEX_LENGTH_BYTES);
		const key = Buffer.concat([
			utils.hash(this._topics[i]).slice(0,
EVENT_TOPIC_HASH_LENGTH_BYTES),
			indexBytes,
		]);
		result.push({
			key,
			value,
		});
	}
	return result;
}
```

Figure 32.1: The method used in calculations of sparse Merkle tree roots (`lisk-sdk/elements/lisk-chain/src/event.ts#71–89`)

Exploit Scenario

An attacker constructs a transaction that triggers a large number of events but is still within the limits defined in LIP-0065. The attacker then constructs a block with the malicious transaction. The block is processed by other nodes, which crash. The chain halts.

Recommendations

Short term, in the `keyPair` method, have the code convert the `indexBit` variable to an unsigned integer before calling the `writeUIntBE` method.

Long term, write tests that will verify correctness of the code for edge cases where values are near the limits defined in various constants.

33. Lack of bounds on reward configuration could cause node crashes	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-33
Target: `lisk-sdk/framework/src/modules/reward/calculate_reward.ts`	

Description

The offset configuration value of the `reward` module can be misconfigured to prevent the node from processing any transactions.

The `calculateDefaultReward` method defined in the `reward` module features a subtraction of the configuration offset from the current height. If the `config.offset` value is set to a value greater than the `height`, the subtraction highlighted in figure 33.1 will yield a negative number. The subsequent ternary check will use this negative number as its `bracket` value and access an out-of-range element of the `brackets` array, returning `undefined`.

```
export const calculateDefaultReward = (config: ModuleConfig, height: number): bigint
=> {
	if (height < config.offset) {
		return BigInt(0);
	}

	const rewardDistance = Math.floor(config.distance);
	const location = Math.trunc((height - config.offset) / rewardDistance);
	const lastBracket = config.brackets[config.brackets.length - 1];

	const bracket =
		location > config.brackets.length - 1 ?
config.brackets.lastIndexOf(lastBracket) : location;

	return config.brackets[bracket];
};
```

Figure 33.1: Subtraction could yield a negative value if the node is misconfigured.
(`lisk-sdk/framework/src/modules/reward/calculate_reward.ts#L17-L27`)

This method is called in various locations throughout the `dynamic_rewards` module, often followed by multiplication by a `BigInt`, resulting in type errors being thrown—for example, in the `getMinimalRewardActiveValidators` method shown in figure 33.2.

```
export const getMinimalRewardActiveValidators = (
	moduleConfig: ModuleConfig,
	defaultReward: bigint,
) =>
	(defaultReward * BigInt(moduleConfig.factorMinimumRewardActiveValidators)) /
	DECIMAL_PERCENT_FACTOR;
```

Figure 33.2: Length checks are missing during validator registration.
(`lisk-sdk/framework/src/modules/dynamic_rewards/utils.ts#L19-L24`)

The `getMinimalRewardActiveValidators` method is called from the `beforeTransactionExecute` method of the `dynamic_rewards` module. If the `defaultReward` value passed to this method is `undefined`, this method will crash with a type error.

Exploit Scenario

A Lisk node administrator accidentally misconfigures her instance and provides an offset that is too large. She receives no warning during node startup, and later, when her node starts processing transactions, it crashes.

Recommendations

Short term, expand the ternary check so that in addition to rounding too large values down to the last item of the bracket, it also rounds too small values up to the first item of the bracket. Alternatively, have the code verify the configuration values during module initialization so that node operators get early warning of a misconfiguration.

Long term, architect the system to prevent accidental misuse. If node operators are capable of sabotaging themselves, some portion of them will.

34. Mainchain registration schema validates signature length incorrectly	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-34
Target: `lisk-sdk/framework/src/modules/interoperability/schemas.ts`	

Description

The `RegisterMainchainCommand` class of the `interoperability` module uses the `mainchainRegParams` schema to validate several parameters, including a `signature` parameter, which is of the `bytes` type. The schema sets the `minItems` and `maxItems` properties for this parameter, as shown in figure 34.1.

```
178    signature: {
179          dataType: 'bytes',
180          fieldNumber: 5,
181          minItems: BLS_SIGNATURE_LENGTH,
182          maxItems: BLS_SIGNATURE_LENGTH,
```

Figure 34.1: The `mainchainRegParams` schema sets the `minItems` and `maxItems` properties for the `signature` parameter.
(`lisk-sdk/framework/src/modules/interoperability/schemas.ts#178–182`)

However, `minItems` and `maxItems` are meant to verify the number of elements in an array, not the length of a byte sequence. As a result, `RegisterMainchainCommand` does not validate the length of the `signature` parameter.

This problem can be detected with the following Semgrep rule.

```
rules:
  - id: schema_min_max_items_without_array
    message: Schema object contains the minItems and/or maxItems properties but is
not of type array
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-either:
            - pattern-inside: >
                {
                    minItems: ...,
                }
            - pattern-inside: >
                {
                    maxItems: ...,
```

```
            }
      - pattern-not-inside: >
          {
              type: 'array',
          }
      - pattern-not-inside: >
          {
              ...$X
          }
```

Figure 34.2: Semgrep rule that identifies schemas with the `minItems` or `maxItems` properties in an object that is not an array

Recommendations

Short term, have the code use the `minLength` and `maxLength` fields to validate the length of the `signature` parameter.

Long term, add a check in the validator to ensure that the `minItems` and `maxItems` properties are set only on objects of the `array` type. Alternatively, integrate the Semgrep rule from figure 34.2 in the CI/CD pipeline. However, this rule may have false negatives (e.g., a schema object where the properties come from a `...var` expansion), so the former recommendation is preferred.

35. Sidechain terminated command uses the wrong chain ID variable

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-35
Target: `lisk-sdk/framework/src/modules/interoperability/sidechain/cc_commands/sidechain_terminated.ts`	

Description

The `execute` function of the `SidechainCCSidechainTerminatedCommand` command incorrectly uses the `context.chainID` variable where it should use the `context.params.chainID` variable, as shown in figure 35.1.

```
public async execute(context: CCCommandExecuteContext<CCMSidechainTerminatedParams>
): Promise<void> {
	const terminatedStateSubstore = this.stores.get(TerminatedStateStore);
	const terminatedStateAccountExists = await terminatedStateSubstore.has(
		context,
		context.chainID,
	);
	if (terminatedStateAccountExists) {
		const terminatedStateAccount = await
terminatedStateSubstore.get(context, context.chainID);
		if (terminatedStateAccount.initialized) {
			return;
		}
		await terminatedStateSubstore.set(context, context.chainID, {
			stateRoot: context.params.stateRoot,
			mainchainStateRoot: EMPTY_HASH,
			initialized: true,
		});
	} else {
		await this.internalMethods.createTerminatedStateAccount(
			context,
			context.params.chainID,
			context.params.stateRoot,
		);
	}
}
```

Figure 35.1: *`lisk-sdk/framework/src/modules/interoperability/sidechain/cc_commands/sidechain_terminated.ts#50–77`*

The `context.chainID` variable contains the `chainID` of the receiving chain (as populated in `lisk-sdk/framework/src/application.ts#L306`), while the `context.params.chainID` variable contains the ID of the sidechain that should be terminated. Since a sidechain's own account will not be in the `TerminatedStateStore`, the `if` check `if (terminatedStateAccountExists)` will never be taken. Instead, the `createTerminatedStateAccount` method will always be called even if the account is already terminated.

Recommendations

Short term, use the `context.params.chainID` variable instead of `context.chainID`.

36. Hex format validator allows empty and odd-length strings	
Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-36
Target: `lisk-sdk/elements/lisk-validator/src/validation.ts`	

Description

The `lisk-validator` method that validates hex-encoded data, `isHexString`, returns `true` if the provided string is empty `['']` or consists of just one hex character (e.g., `f`), as shown in line 77 of figure 36.1.

```
72    export const isHexString = (data: unknown): boolean => {
73          if (typeof data !== 'string') {
74                return false;
75          }
76
77          return data === '' || /^[a-f0-9]+$/i.test(data);
78    };
```

Figure 36.1: `lisk-sdk/elements/lisk-validator/src/validation.ts#72–78`

Both cases cause a subsequent call to `Buffer.from(data, 'hex')` to return an empty `Buffer` object. This may violate the assumption that because the string passed hex format validation, it must encode usable data. Furthermore, when the number of characters is odd but greater than one (e.g., `fff`), it results in incomplete decoding.

Recommendations

Short term, modify `isHexString` to reject zero-length and odd-length strings.

Long term, write a simple fuzzing harness that attempts to identify invalid hex strings that pass the `isHexString` check.

37. CCM fees are always burned	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-37
Target: `lisk-sdk/framework/src/modules/fee/cc_method.ts`	

Description

The `fee` module's `afterCrossChainCommandExecute` function—which is responsible for managing the consumed fee after the execution of a CCM—can handle the consumed fee, according to the LIP: it may burn the tokens or transfer them to an `ADDRESS_FEE_POOL` variable (figure 37.1). However, the implementation always burns the used fee (figure 37.2).

```
burnConsumedFee = False if (ADDRESS_FEE_POOL is not None and
Token.userSubstoreExists(ADDRESS_FEE_POOL, messageFeeTokenID)) else True

if burnConsumedFee:
    Token.burn(relayerAddress, messageFeeTokenID, ccm.fee - ctx.availableCCMFee)
else:
    # Transfer the cross-chain message fees to the pool account address, if it
exists and it is initialized for the cross-chain message fee token.
    Token.transfer(relayerAddress, ADDRESS_FEE_POOL, messageFeeTokenID, ccm.fee -
ctx.availableCCMFee)
```

Figure 37.1: Part of the pseudo code of the `afterCrossChainCommandExecute` function as defined in LIP-0048 (lips/proposals/lip-0048.md#after-cross-chain-command-execution)

```
const burntAmount = ctx.ccm.fee - availableFee;
await this._tokenMethod.burn(
       ctx.getMethodContext(),
       ctx.transaction.senderAddress,
       messageTokenID,
       burntAmount,
);
```

Figure 37.2: Part of the implementation of the `afterCrossChainCommandExecute` function in the LIP (`lisk-sdk/framework/src/modules/fee/cc_method.ts#73–79`)

Recommendations

Short term, fix the implementation so that it matches the LIP. If the implementation is knowingly incomplete and therefore intentionally does not match the LIP, add a comment to the code mentioning it. This will make it clear for anyone auditing the code and will reduce the likelihood of releasing the incomplete implementation.

38. CCM fees are underspecified and the implementation is not defensive

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-38
Target: `lisk-sdk`, CCM fees	

Description

The CCM fee mechanism is hard to fully understand from the LIPs because the functions that handle CCM fees are specified across several LIPs, without a single location where the entire concept is explained. Furthermore, the implementation is spread over the following locations in the code, which makes it harder to connect all the dots:

- The `payMessageFee` function in the `token` module
- The `verifyCrossChainMessage` and `beforeCrossChainCommandExecute` CCMs in the `token` module
- The `beforeCrossChainCommandExecute` and `afterCrossChainCommandExecute` CCMs in the `fee` module
- The `bounce` method of the `interoperability` module

The CCM fee implementation also includes generalizations that would enable future modification to the protocol, such as the `messageFeeTokenID` property, which specifies the token to be used when paying fees. Currently, this property is always LSK.

The implementation is not defensive when handling the `messageFeeTokenID` property for multiple reasons.

First, the `beforeCrossChainMessageForwarding` function assumes that `getMessageFeeTokenID` will always return LSK. From discussions with the Lisk team, the plans are for channels between sidechains and the mainchain to always use LSK tokens to pay for CCM fees. However, if this ever changes, the current implementation would be incorrect because it tries to modify escrow amounts of a potentially non-native token. Most likely this attempt would crash and revert the transaction, but, for example, a fee of `0` would create an entry in the escrow store for a non-native token, which should never happen.

```
public async beforeCrossChainMessageForwarding(ctx: CrossChainMessageContext):
Promise<void> {
```

```
        const { ccm } = ctx;
        const methodContext = ctx.getMethodContext();
        const messageFeeTokenID = await this._interopMethod.getMessageFeeTokenID(
                methodContext,
                ccm.receivingChainID,
        );
        const { ccmID } = getEncodedCCMAndID(ccm);

        const escrowStore = this.stores.get(EscrowStore);
        const escrowKey = escrowStore.getKey(ccm.sendingChainID, messageFeeTokenID);
        const escrowAccount = await escrowStore.getOrDefault(methodContext,
escrowKey);
        if (escrowAccount.amount < ccm.fee) {
                [REDACTED]
        }

        escrowAccount.amount -= ccm.fee;
        await escrowStore.set(methodContext, escrowKey, escrowAccount);

        await escrowStore.addAmount(methodContext, ccm.receivingChainID,
messageFeeTokenID, ccm.fee);
```

Figure 38.1: `lisk-sdk/framework/src/modules/token/cc_method.ts#83–114`

Secondly, the code uses `getMessageFeeTokenID(ccm.receivingChainID)` and `getMessageFeeTokenID(ccm.sendingChainID)` interchangeably. For example, the `beforeCrossChainCommandExecute` function uses `sendingChainID` (contrary to what is specified in the LIP), while `beforeCrossChainMessageForwarding` uses `receivingChainID`. From discussions with the Lisk team, the `messageFeeTokenID` of a channel between two chains will always be the same, so both calls would always return the same value. However, if future code ever allows each end of the channel to use a different fee token, the code could be vulnerable.

Exploit Scenario

Due to the lack of specification and defensive code, implementation of an extension of the protocol with different assumptions than the current ones results in a vulnerability. An attacker exploits this vulnerability, which results in the loss of funds.

Recommendations

Short term, if the `beforeCrossChainMessageForwarding` function of the `token` module should work only with LSK tokens, add a check to ensure that `messageFeeTokenID` is always LSK. Create a function that, given a CCM, returns the `messageFeeTokenID`. This will reduce the ambiguity of using `receivingChainID` or `sendingChainID` and ensure that all functions use a consistent value.

Long term, improve the specification of CCM fee handling. This will help users and developers understand how CCM fees should work.

39. Unspecified order for running interoperability modules	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-LISK-39
Target: LIP-0063, `lisk-sdk`	

Description

The order in which `interoperability` modules are registered—which dictates the order in which functions such as `verifyCrossChainMessage` and `beforeCrossChainCommandExecute` are executed—is not specified in LIP-0063.

The order of modules in normal transactions is specified in LIP-0063, and as exemplified by this issue that describes how this order was incorrectly implemented, this specification is fundamental for the correct functioning of features such as fees.

Recommendations

Short term, in LIP-0063, specify the order in which `interoperability` modules should be registered, including the reasoning for the decided ordering.

40. The interoperability module's terminateChain method does not work

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-LISK-40
Target: `lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts`	

Description

The `terminateChain` method of the `interoperability` module always fails. The `terminateChain` method uses the `getTerminatedStateAccount` function to determine if the chain is already terminated, in which case it returns early (highlighted in red in figure 40.2). If `getTerminatedStateAccount` returns a falsy value (e.g., `undefined`), then `terminateChain` terminates the chain.

However, `getTerminatedStateAccount` throws a `NotFoundError` exception when the chain is not present in the `TerminatedStateStore`, which causes the `terminateChain` method to bubble up the exception, so `sendInternal` is never called.

```
public async getTerminatedStateAccount(context: ImmutableMethodContext, chainID: Buffer) {
	return this.stores.get(TerminatedStateStore).get(context, chainID);
}
```

Figure 40.1: The `getTerminatedStateAccount` function (`lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts#87–89`)

```
// https://github.com/LiskHQ/lips/blob/main/proposals/lip-0045.md#terminatechain
public async terminateChain(context: MethodContext, chainID: Buffer): Promise<void>
{
	if (await this.getTerminatedStateAccount(context, chainID)) {
		return;
	}

	await this.internalMethod.sendInternal(
		context,
		EMPTY_FEE_ADDRESS,
		MODULE_NAME_INTEROPERABILITY,
		CROSS_CHAIN_COMMAND_CHANNEL_TERMINATED,
		chainID,
		BigInt(0),
		CCMStatusCode.OK,
```

```
            EMPTY_BYTES,
        );

        await this.internalMethod.createTerminatedStateAccount(context, chainID);
}
```

Figure 40.2: The `terminateChain` method (`lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts#168–186`)

The existing test cases do not catch this issue because `getTerminatedStateAccount` is mocked in a way that does not reflect the correct behavior of that function, as shown in figure 40.3.

```
jest.spyOn(sampleInteroperabilityMethod as any,
'getTerminatedStateAccount').mockResolvedValue(undefined);

await sampleInteroperabilityMethod.terminateChain(methodContext, chainID);
```

Figure 40.3: Test where the `getTerminatedStateAccount` function is mocked to return undefined (`lisk-sdk/framework/test/unit/modules/interoperability/method.spec.ts#520–524`)

Exploit Scenario

A module developer relies on the `terminateChain` method to prevent malicious behavior. The `terminateChain` method does not terminate the chain, which breaks the developer's assumptions and allows the malicious chain to continue misusing the honest chain.

Recommendations

Short term, fix the `terminateChain` method by using a try-catch construct around the `getTerminatedStateAccount` function.

Long term, consider writing a CodeQL rule that catches cases where the result of a `store.get` method is used directly in an `if` statement. This is likely to indicate the presence of a bug.

41. Chain ID length not validated in interoperability endpoints	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-41
Target: `lisk-sdk/framework/src/modules/interoperability/schemas.ts`, `lisk-sdk/framework/src/modules/interoperability/base_interoperability_endpoint.ts`	

Description

The `BaseInteroperabilityEndpoint` and `MainchainInteroperabilityEndpoint` classes provide several methods that take a hex-formatted chain ID as a parameter. These methods validate the chain ID using different schemas that are all aliases for the `getChainAccountRequestSchema` schema (figure 41.1).

```
export const getChainAccountRequestSchema = {
	$id: '/modules/interoperability/endpoint/getChainAccountRequest',
	type: 'object',
	required: ['chainID'],
	properties: {
		chainID: {
			type: 'string',
			format: 'hex',
		},
	},
};
```

Figure 41.1: The `getChainAccountRequestSchema` schema (`lisk-sdk/framework/src/modules/interoperability/schemas.ts#561–571`)

The `chainID` property in the above schema lacks the `minLength` and `maxLength` fields. As a result, arbitrary-length chain ID strings may be passed to internal functions that assume a correct length.

Recommendations

Short term, set the `minLength` and `maxLength` fields for the `chainID` property of `getChainAccountRequestSchema` to `CHAIN_ID_LENGTH` * 2 (to take hex encoding into account).

42. Bounced CCM fees are not escrowed	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-42
Target: `lisk-sdk, Cross-Chain Messages Fees`	

Description

The fee of bounced messages is not correctly processed. Bounced messages are cross-chain messages (CCMs) that are returned to the sending chain when an error occurs during their processing in the target chain. The Lisk protocol lightly penalizes the sending chain for the bad CCM that needed to be bounced, but the sending chain can recover most of the fee (since no real processing occurred on the target chain).

If, for example, the target chain does not support a given module, the cross-chain update (CCU) relayer receives no fee, but the fee of the bounced CCM is reduced by the `minFee` value to penalize the chain that sent the bad CCM (the relayer of the newly bounced CCM can get only part of the fee that originated from the sending chain).

A problem occurs when subtracting `minFee` because the receiving chain does not burn it or update the other sidechain's escrow balance. This means that the sidechain can recover the full fee amount, circumventing the penalization for the bad CCM (the regular fee for each byte of the CCM is still paid by the relayer when submitting the CCU transaction).

This issue is present in the `apply` method, when a module is not supported or when a command in a module is not supported, and in the `forward` method, when the mainchain does not have a channel to the receiving chain or when this channel is only in the registered state. Transactions that fail on the `execute` method are not affected because, in that case, the fee has already been paid to the relayer.

Exploit Scenario

A sidechain sends a CCM to the mainchain, targeting the `ABC` module with a fee of 10 LSK. The mainchain does not support the `ABC` module, which triggers a bounce with a fee of 8 LSK after subtracting the `minFee` of 2 LSK for the bad transaction. The mainchain does *not* update the escrow balance of the sidechain. The sidechain receives the `bounced` request with the updated, smaller fee of 8 LSK. The sidechain retrieves all 10 LSK since the escrow amount was not updated.

Recommendations

Long term, create a cohesive specification describing fee handling for both normal transactions and CCMs. Include the detailed rationale for why the solution was chosen over other alternatives.

The current lack of a holistic specification and having the implementation spread over several modules (the `fee` module, the `token` module, and the `bounce` method of the `interoperability` module) makes it harder to understand fees. This spread-out implementation also makes it harder for a module developer to implement new fee mechanisms without modifying all three modules (`fee`, `token`, and `interoperability`); ideally a developer should need to modify only the `fee` module, which would require exposing more methods from the `token` module, such as `addAvailableBalanceAndUpdateEscrow` (an alternative name could be `receiveBalanceFromChain`), and the `swapEscrowBalance` method on the mainchain to handle forwarded messages.

This behavior exposes more internals of the `token` module, which might be unwanted; however, because of the tight coupling of the `fee` and `token` modules, it seems hard to implement all fee functionality in the `fee` module without exposing more internals.

43. The send method accepts a status different from OK	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-43
Target: `lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts`	

Description

LIP-0045 explains that the send method of the `interoperability` module always calls the `sendInternal` internal method with a status of OK. However, the implementation of send receives the status as a parameter, as shown in figure 43.1. This means that another module may use the send method with an arbitrary status, including error statuses that are reserved.

```
public async send(
	context: MethodContext,
	sendingAddress: Buffer,
	module: string,
	crossChainCommand: string,
	receivingChainID: Buffer,
	fee: bigint,
	status: number,
	params: Buffer,
	timestamp?: number,
): Promise<void> {
	await this.internalMethod.sendInternal(
		context,
		sendingAddress,
		module,
		crossChainCommand,
		receivingChainID,
		fee,
		status,
		params,
		timestamp,
	);
}
```

Figure 43.1: *`lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts#125–147`*

Exploit Scenario

The `interoperability` module creates assumptions based on the `send` method always using the `OK` status. A module uses the `send` method with a reserved error status, breaking this assumption and causing other issues.

Recommendations

Short term, rewrite the `send` method to not receive the `status` parameter. Instead, have the code call `sendInternal` with a hard-coded status of `OK`, as specified in the LIP.

44. The error method incorrectly checks the status field

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-44
Target: `lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts`	

Description

The `error` method of the `interoperability` module incorrectly verifies the bounds of its `status` parameter. As specified in the LIP, the `error` method attempts to guarantee that the error status of module-initiated errors is greater than the `MAX_RESERVED_ERROR_STATUS` constant; however, with the check highlighted in the figure below, it will error out only on negative numbers and zero.

```
public async error(context: MethodContext, ccm: CCMsg, errorStatus: number):
Promise<void> {
	// Error codes from 0 to MAX_RESERVED_ERROR_STATUS (included) are reserved to
the Interoperability module.
	if (errorStatus <= 0 && errorStatus <= MAX_RESERVED_ERROR_STATUS) {
		throw new Error('Invalid error status.');
	}

	await this.send(
		context,
		EMPTY_FEE_ADDRESS,
		ccm.module,
		ccm.crossChainCommand,
		ccm.sendingChainID,
		BigInt(0),
		errorStatus,
		ccm.params,
	);
}
```

Figure 44.1: *`lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts#150–166`*

Exploit Scenario

A custom module of a sidechain contains a bug where a user can control the value of the status flowing to the `error` method during a transaction. An attacker sets this value to a reserved error code, breaking assumptions of the `interoperability` module and potentially leading to a termination of the sidechain.

Recommendations

Short term, modify the check to ensure that values less than or equal to `MAX_RESERVED_ERROR_STATUS` error out.

Long term, add tests to the `error` method to prevent regressions.

45. Send method may lead to crash when missing the timestamp parameter

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-45
Target: `lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts`	

Description

The send method of the `interoperability` module receives an optional `timestamp` parameter (figure 45.1), which is not specified in the LIP and may lead to crashes when not set.

```
public async send(
	context: MethodContext,
	sendingAddress: Buffer,
	module: string,
	crossChainCommand: string,
	receivingChainID: Buffer,
	fee: bigint,
	status: number,
	params: Buffer,
	timestamp?: number,
): Promise<void> {
	await this.internalMethod.sendInternal(
		context,
		sendingAddress,
		module,
		crossChainCommand,
		receivingChainID,
		fee,
		status,
		params,
		timestamp,
	);
}
```

Figure 45.1: *`lisk-sdk/framework/src/modules/interoperability/base_interoperability_method.ts#125–147`*

On send methods originating from the mainchain, the `timestamp` parameter is used to check the liveness of a chain with the `isLive` function (sidechains do not check the liveliness of other chains). So, if this optional parameter is not set when calling send on the mainchain, the code will throw an exception. This is what happens with the `transferCrossChain` method of the `token` module, which does not pass a `timestamp` to the `send` method. If the mainchain uses the `transferCrossChain` method, the `send` method will unconditionally fail because the `isLive` function throws an exception when trying to use the undefined `timestamp`.

Recommendations

Short term, fix the `transferCrossChain` method by having it pass the `timestamp` parameter (stored inside the `methodContext` argument) to the `send` method.

Long term, update the LIP to reflect the implementation's use of the `timestamp` parameter. To avoid similar problems in the future, consider making the `timestamp` parameter required instead of optional.

46. The channel terminated command does not validate the CCM status	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-46
Target: `lisk-sdk/framework/src/modules/interoperability/base_cc_commands/channel_terminated.ts`	

Description

The `BaseCCChannelTerminatedCommand` command does not validate the status of the incoming cross-chain message (CCM). As a result, if a CCM of this type is bounced, the command will try to add the sending chain that bounced the request to the `TerminatedStateStore` store.

In this command, this issue does not pose a major problem. If the chain sends this CCM to the target chain, it likely already terminated the target chain in the first place. However, in other cases, not verifying the status may be problematic. For example, a bounced message to the `token` module could result in double spending if the CCM status was not checked.

Recommendations

Short term, have the code check the status of the CCM in the `BaseCCChannelTerminatedCommand`'s `verify` function, and process the request only if the status is `OK`.

Long term, consider improving the handling of bounced CCMs. When an error method is returned, consider having the code call a different function than `verify` and `execute`, which are used for CCMs with a status of `OK`. Instead, new functions such as `verifyOnError` and `executeOnError` could be created to handle error CCMs (also consider having just `executeOnError` but keeping the `verify` function the same for both). These changes would make error handling clearer and remove the burden on developers to remember to verify the `status` field on every cross-chain command.

47. Invalid use of values in the sparse Merkle tree	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-47
Target: `lisk-sdk`, `InitializeMessageRecoveryCommand`	

Description

LIP-0039 allows users to pass arbitrarily sized byte arrays as values to the sparse Merkle tree (SMT) API. However, as observed in the codebase, SMT users usually pass hashes of values instead of raw values. This behavior introduces an ambiguity in how the SMT should be used.

First of all, using hashes as values is a safe mechanism but only if applied consistently. If there is a code path where a raw value is accepted, then the value may collide with a hash of some other value that is set in an SMT. In the worst case, it would be possible to construct malicious inclusion or noninclusion proofs.

Moreover, setting a value to an empty one—that is, removing a key-value pair from an SMT—is an underspecified operation. Developers are not provided with a clear specification about how to remove a key-value pair. For example, in the `execute` function of the `BaseStateRecoveryCommand` command, an empty hash (a hash of an empty string) is used to remove a key-value pair; however, this is invalid. Instead of removing the key-value pair, the value is changed but still exists, so an inclusion proof can be constructed for the key, and the value can be set to empty bytes.

```
storeQueries.push({
	key: Buffer.concat([storePrefix, entry.substorePrefix,
utils.hash(entry.storeKey)]),
	value: EMPTY_HASH,
	bitmap: entry.bitmap,
});
```

Figure 47.1: Part of the `BaseStateRecoveryCommand` command (`lisk-sdk/framework/src/modules/interoperability/base_state_recovery.ts#165–169`)

One more ambiguity exists in the `InitializeMessageRecoveryCommand` command shown in figure 47.2. The `params.channel` variable is hashed before being used as a value in the SMT but is not validated to be nonempty. It is unclear if providing empty bytes as the `params.channel` variable should constitute a noninclusion proof, an inclusion proof, or an inclusion proof for an incorrectly removed value (as mentioned in the previous paragraph).

```
const query = {
	key: queryKey,
	value: utils.hash(params.channel),
	bitmap: params.bitmap,
};

const smt = new SparseMerkleTree();
const valid = await smt.verify(terminatedAccount.stateRoot, [queryKey], {
	siblingHashes: params.siblingHashes,
	queries: [query],
});
```

Figure 47.2: Code where the value of a query passed to the `verify` function may be the hash of the empty string, resulting in an ambiguous proof (`lisk-sdk/framework/src/modules/interoperability/mainchain/commands/initialize_message_recovery.ts#112–122`)

Exploit Scenario

An attacker constructs a valid `BaseStateRecoveryCommand` command for some state, represented as a key-value pair, that exists in the system and can be recovered honestly. She sends that command to the chain, which recovers the state and incorrectly updates the `terminatedStateSubstore` SMT.

The attacker constructs a new `BaseStateRecoveryCommand` for the same state (resulting in the same key), but with an empty value. The chain accepts the new command as valid and executes the recovery for the state again. Because the attacker cannot control the value, the impact of the attack is low. However, if the recovery method changes the chain's state, even for invalid or empty values, the impact will be high.

Recommendations

Short term, if passing hashes instead of raw values is the intended way of using the SMT, document this assumption in the relevant LIP and ensure the SMT implementation enforces the use of hashes. For example, add a check to ensure that lengths of provided values are equal to an expected constant (i.e., to the size of outputs from the hashing function). Alternatively, consider using raw values in the SMT. Validate that the `params.channel` variable is not the empty string. This will ensure that the `verify` function always performs an inclusion proof.

Long term, modify the SMT's `verify` function API to have a clear distinction between inclusion and noninclusion proofs. To prevent consumers of the SMT API from incorrectly using the API, consider rejecting empty values and/or hashes of empty values in the inclusion or noninclusion verification methods.

48. Recovered messages can be replayed	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-48
Target: `lisk-sdk`, message recovery functionality	

Description

The message recovery functionality does not prevent recoverable messages from being replayed. This should be prevented with the following security controls:

- In the `execute` function of the `RecoverMessageCommand` class, update the `status` of each recovered CCM to `CCMStatusCode.RECOVERED`, and update the `outboxRoot` of the terminated outbox account with this new value, as shown in figure 48.1.
- In the `verify` function of the `RecoverMessageCommand` class, prevent CCMs with a `status` other than `CCMStatusCode.OK` from being processed, as shown in figure 48.2.

```
// Update sidechain outbox root.
const proof = {
       size: terminatedOutboxAccount.outboxSize,
       indexes: params.idxs,
       siblingHashes: params.siblingHashes,
};

terminatedOutboxAccount.outboxRoot = regularMerkleTree.calculateRootFromUpdateData(
       recoveredCCMs.map(ccm => utils.hash(ccm)),
       proof,
);
```

Figure 48.1: The `execute` method updating the terminated account's `outboxRoot` with a CCM with an updated `status` of `CCMStatusCode.RECOVERED`
(`lisk-sdk/framework/src/modules/interoperability/mainchain/commands/recover_message.ts#217–227`)

```
if (ccm.status !== CCMStatusCode.OK) {
	return {
		status: VerifyStatus.FAIL,
		error: new Error('Cross-chain message status is not valid.'),
	};
}
```

Figure 48.2: The `verify` function checking that a CCM trying to be recovered has a status of `CCMStatusCode.OK` (*`lisk-sdk/framework/src/modules/interoperability/mainchain/commands/recover_message.ts#127–132`*)

Two problems exist that prevent this security control from working. First, the code appends `crossChainMessage` to the `recoveredCCMs` array. `crossChainMessage` is the originally encoded CCM and not the CCM updated by the `_applyRecovery` and `_forwardRecovery` functions, as shown in figure 48.3.

```
for (const crossChainMessage of params.crossChainMessages) {
	const ccmID = utils.hash(crossChainMessage);
	const ccm = codec.decode<CCMsg>(ccmSchema, crossChainMessage);
	const ctx: CrossChainMessageContext = {
		...context,
		ccm,
		eventQueue: context.eventQueue.getChildQueue(ccmID),
	};
	// If the sending chain is the mainchain, recover the CCM.
	// This function never raises an error.
	if (ccm.sendingChainID.equals(getMainchainID(context.chainID))) {
		await this._applyRecovery(ctx);
	} else {
		// If the sending chain is not the mainchain, forward the CCM.
		// This function never raises an error.
		await this._forwardRecovery(ctx);

		// Append the recovered CCM to the list of recovered CCMs.
		// Notice that the ccm has been mutated in the applyRecovery and
forwardRecovery functions
		// as the status is set to CCM_STATUS_CODE_RECOVERED (so that it cannot
be recovered again).
		recoveredCCMs.push(crossChainMessage);
	}
}
```

Figure 48.3: The incorrect value is appended to the `recoveredCCM` array; the append also happens in the `else` branch, which is incorrect. (*`lisk-sdk/framework/src/modules/interoperability/mainchain/commands/recover_message.ts#187–209`*)

Secondly, the CCM is not updated in the `_forwardRecovery` and `_applyRecovery` functions, which solely create the `recoveredCCM` variable without updating the CCM stored in the `context` object, as shown in figure 48.4.

```
// eslint-disable-next-line @typescript-eslint/ban-ts-comment
private async _applyRecovery(context: CrossChainMessageContext): Promise<void> {
	const { logger } = context;
	const { ccmID } = getEncodedCCMAndID(context.ccm);
	const recoveredCCM: CCMsg = {
		...context.ccm,
		status: CCMStatusCode.RECOVERED,
		sendingChainID: context.ccm.receivingChainID,
		receivingChainID: context.ccm.sendingChainID,
	};
```

Figure 48.4: Snippet showing that `context.ccm` is not updated (`lisk-sdk/framework/src/modules/interoperability/mainchain/commands/recover_message.ts#232–241`)

Both of these flaws independently cause the `recoveredCCM` array to have the original CCM, leaving the `outboxRoot` unaltered and allowing the CCM to be recovered repeatedly. Furthermore, CCMs targeting the mainchain can never be recovered because the code that appends the CCM to the `recoveredCCMs` array executes only in the `else` block, as shown in figure 48.3. This is incorrect and different from the LIP's pseudocode shown in figure 48.5, where the `recoveredCCM` is appended independently of the branch taken. This makes messages targeting the mainchain irrecoverable because the `calculateRootFromUpdateData` function (shown in figure 48.1) will fail when the `recoveredCCM` array has a different length than the `params.idxs` array.

```
# If the sending chain is the mainchain, recover the CCM.
# This function never raises an error.
if ccm.sendingChainID == getMainchainID():
    applyRecovery(trs, ccm)
# If the sending chain is not the mainchain, forward the CCM.
# This function never raises an error.
elif ccm.sendingChainID != getMainchainID():
    forwardRecovery(trs, ccm)

# Append the recovered CCM to the list of recovered CCMs.
# Notice that the ccm has been mutated in the applyRecovery and forwardRecovery
functions
# as the status is set to CCM_STATUS_CODE_RECOVERED (so that it cannot be recovered
again).
recoveredCCMs.append(encode(crossChainMessageSchema, ccm))
```

Figure 48.5: LIP pseudo-code that shows how the code should prevent recovered CCMs from being replayed (`https://github.com/LiskHQ/lips/blob/main/proposals/lip-0054.md#execution-1`)

Exploit Scenario

An attacker finds a vulnerability that allows him to terminate a sidechain—for example, by finding an input that causes the `verify` function of a custom CCM of the sidechain to fail. The attacker sends several cross-chain token transfers to that sidechain just before terminating it. At least one of these cross-chain transfers will not be updated in the sidechain, so it will be a recoverable message. The attacker recovers the cross-chain token transfer message multiple times until all the sidechain's escrow is exhausted. The attacker effectively recovers all the sidechain tokens, including tokens that do not belong to them, which prevents other users from recovering their tokens.

Recommendations

Short term, fix the immediate vulnerabilities by modifying the `_applyRecovery` and `_forwardRecovery` functions to update the CCM in the `context` object and then encoding and appending the updated CCM to the `recoveredCCM` array (instead of appending the encoding of the old CCM) outside the `else` branch. This will prevent recoverable CCMs from being replayed.

Long term, improve testing to prevent issues such as these from being introduced or reintroduced. Currently, the use of `jest.spyOn(regularMerkleTree, 'verifyDataBlock').mockReturnValue(true)` makes it harder to unit test invariants, such as that a CCM should be recoverable only once. In addition to these unit tests, create a list of invariants that should always hold, and test them in end-to-end tests on a running blockchain system with a mainchain and several sidechains. If possible, use fuzzing to test these invariants thoroughly.

49. StateStore handles multiple snapshots dangerously	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-49
Target: `lisk-sdk/elements/lisk-chain/src/state_store/state_store.ts`	

Description

The `StateStore` class supports only one snapshot at a time. Creating two snapshots and then trying to recover the first one (by passing the corresponding `id` to the `restoreSnapshot` function) will result in recovering the latest snapshot, not the first one. As shown in figure 49.1, the `createSnapshot` function will always return an `id` of `0`, and the `restoreSnapshot` function will not use the `id` at all.

```
// createSnapshot follows the same interface as stateDB. However, it does not
support multi snapshot.
public createSnapshot(): number {
	this._snapshot = this._cache.copy();
	return 0;
}

// restoreSnapshot does not support multi-snapshot. Therefore, id is not used.
public restoreSnapshot(_id: number): void {
	if (!this._snapshot) {
		throw new Error('Snapshot must be taken first before reverting');
	}
	this._cache = this._snapshot;
	this._snapshot = undefined;
}
```

Figure 49.1: The `StateStore` class's `createSnapshot` and `restoreSnapshot` functions (`lisk-sdk/elements/lisk-chain/src/state_store/state_store.ts#222–235`)

The severity of this finding is set to informational because we did not find functions that call the `createSnapshot` or `restoreSnapshot` functions of the `StateStore` class. The `StateMachine` class and the `apply` and `forward` functions of the `interoperability` module use `createSnapshot` or `restoreSnapshot` on the `ctx.stateStore` variable; however, this variable is an instance of the `PrefixedStateReadWriter` class, which supports multi-snapshots.

Exploit Scenario

The code is modified to use the snapshot functionality of the `StateStore` class. When recovering from a snapshot that is not the latest (as can happen in the `StateMachine` class and in the `apply` function of the `interoperability` module), the store restores the latest snapshot. This leads to broken assumptions and potential bugs.

Recommendations

Short term, modify the code to return an incrementing `id` in the `StateStore`'s `createSnapshot` function. Then, have the `recoverSnapshot` function throw an error if the `id` received does not match the latest created snapshot. This will allow the class to support only one snapshot at a time while having defensive code that prevents its misuse.

50. Invalid base method used in InitializeStateRecoveryCommand	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-50
Target: `lisk-sdk/framework/src/modules/interoperability/sidechain/commands/initialize_state_recovery.ts`	

Description

The `InitializeStateRecoveryCommand` class extends the `BaseInteroperabilityCommand` class that is instantiated with the `MainchainInteroperabilityInternalMethod` method, instead of the `SidechainInteroperabilityInternalMethod` method. Therefore, the state recovery initialization on a sidechain uses an incorrect set of internal method implementations.

```
export class InitializeStateRecoveryCommand extends
BaseInteroperabilityCommand<MainchainInteroperabilityInternalMethod> {
```

Figure 50.1: The invalid base method (`lisk-sdk/framework/src/modules/interoperability/sidechain/commands/initialize_state_recovery.ts#35`)

The issue is set only to informational severity because only the `isLive` method's implementation depends on the specific instantiation, and the method is currently not used in `InitializeStateRecoveryCommand`.

Recommendations

Short term, replace `MainchainInteroperabilityInternalMethod` with `SidechainInteroperabilityInternalMethod` in the `InitializeStateRecoveryCommand` class definition.

Long term, write a Semgrep rule to test that all relevant classes in the `sidechain` directory are instantiated with `SidechainInteroperabilityInternalMethod` and that all relevant classes in the `mainchain` directory are instantiated with `MainchainInteroperabilityInternalMethod`. Alternatively, research generic constraints and implement them to enforce using only the correct classes.

51. Incorrect handling of large integers in regular Merkle tree verification	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-51
Target: `lisk-sdk/elements/lisk-tree/src/merkle_tree/utils.ts`	

Description

The `messageRecoveryParamsSchema` schema accepts proof indexes of an unsigned, 32-bit integer type. However, large indexes are not handled correctly by the regular Merkle tree, as they are implicitly converted to signed 32-bit integers.

The `messageRecoveryParamsSchema` schema is used by the `RecoverMessageCommand` command, which specifies the `idxs` field (proof indexes) to bytes of type `uint32`. There may be other schemas in the system that use the same type for regular Merkle tree indexes.

```
idxs: {
	type: 'array',
	items: {
		dataType: 'uint32',
	},
	fieldNumber: 3,
},
```

Figure 51.1: Part of the `messageRecoveryParamsSchema` (`lisk-sdk/framework/src/modules/interoperability/schemas.ts#306–312`)

The indexes are validated in the `RecoverMessageCommand`'s `verify` function. The indexes must be unique, sorted in ascending order, and greater than some specific value. A large, positive integer passes all these validations.

Then the indexes are passed as parts of an inclusion proof to the `regularMerkleTree.verifyDataBlock` method, which calls the `calculatePathNodes` method, shown in figure 51.2. In `calculatePathNodes`, an index is right bit-shifted. Because TypeScript's bit operations convert arguments to signed 32-bit integers, a large positive index will result in a negative `parentIdx`. Because the code expects all indexes to be positive, it will behave incorrectly. In the worst case, the bug may allow coercing some indexes during execution of the vulnerable method—that is, a negative index may be treated as a positive index, thus bypassing the uniqueness check from the `verify` function and enabling double-inclusion of data in the tree.

```
const currentHash = tree.get(idx) ?? parentCache.get(idx);
const parentIdx = idx >> 1;
if (!currentHash) {
	throw new Error(`Invalid state. Hash for index ${idx} should exist.`);
}
const currentLoc = getLocation(idx, height);
```

Figure 51.2: Part of the `calculatePathNodes` method (`lisk-sdk/elements/lisk-tree/src/merkle_tree/utils.ts#247–252`)

Exploit Scenario

An attacker forges a proof for `RecoverMessageCommand` with a large index. The proof size—equal to the outbox size—is also large. The index is incorrectly handled by the regular Merkle tree verification code, and the code verifies the proof as valid. The attacker recovers selected messages multiple times.

Recommendations

Short term, have the code either use the unsigned right-shift operator instead of the signed right-shift operator, or validate the indexes to be smaller than the maximum signed 32-bit integer.

Long term, review uses of bitwise operations in the codebase and ensure that all are safe—that is, all operands of such operations are validated to be in a signed 32-bit integer range. Write a lint or a static analysis rule to detect all uses of bitwise operations. Ideally, the rule should distinguish between validated and unvalidated integers, and the CI pipeline should error out when instances of binary operations on unvalidated integers are detected.

52. Lisk Validator does not validate integer formats when provided as number	
Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-52
Target: `lisk-sdk/elements/lisk-validator`	

Description

Lisk Validator supports the `int32`, `int64`, `uint32`, and `uint64` data formats, which can be set in schemas as the value of a property's `format` field. However, we found that specifying one of these formats for a property whose data type is set to `integer` (i.e., it is encoded as a number, as opposed to a string) results in the format not being validated.

For example, the schema for the `setHashOnionUsage` request from the `random` module defines a `height` property with a `type` set to `integer` and a `format` set to `uint32`:

```
export const setHashOnionUsageRequest = {
	$id: 'lisk/random/setHashOnionUsageRequest',
	type: 'object',
	required: ['address', 'count', 'height'],
	properties: {
		address: {
			type: 'string',
			format: 'lisk32',
		},
		count: {
			type: 'integer',
			minimum: 1,
		},
		height: {
			type: 'integer',
			format: 'uint32',
		},
	},
};
```

Figure 52.1: `lisk-sdk/framework/src/modules/random/schemas.ts#129–147`

Despite this, it is possible to provide a value of `height` to `setHashOnionUsage` that is negative or outside the range of a 32-bit unsigned integer:

```
{"jsonrpc": "2.0", "id": 1, "method": "random_setHashOnionUsage",
"params":{"address": "lskkgfk84gngotqjsqfujkctshvkwzs8z8w93gmja", "count": 5,
"height":-1}}
```

Figure 52.2: Providing a negative height to `setHashOnionUsage` succeeds, even though it is defined as a `uint32`.

Furthermore, we created the following `lisk-validator` tests to ensure correct validation of integer formats. The tests fail to reject negative `uint32` and `int32` values above the range of a 32-bit integer, which confirms the issue in the Lisk codebase:

```
import { validator } from '../src';

describe('validator uint32 format', () => {
    describe('integer format validation', () => {
        const uint32SchemaInteger = {
            type: 'object',
            required: ['height'],
            properties: {
                height: {
                    type: 'integer',
                    format: 'uint32',
                },
            },
        };

        // We expect this to throw but it doesn't
        it('should be invalid if negative (number)', () => {
            expect(() =>
                validator.validate(
                    {
                        ...uint32SchemaInteger,
                    },
                    { height: -1 },
                ),
            ).toThrow();
        });

    });

    const int32SchemaInteger = {
        type: 'object',
        required: ['height'],
        properties: {
            height: {
                type: 'integer',
                format: 'int32',
            },
        },
    };

    describe('int32', () => {
```

```
        // We expect this to throw but it doesn't
        it('should be invalid if too big (integer)', () => {
            expect(() =>
                validator.validate(
                    {
                        ...int32SchemaInteger,
                    },
                    { height: 2147483648 },
                ),
            ).toThrow();
        });
    });
});
```

Figure 52.3: Test cases to validate that the `uint32` and `int32` formats fail

The following Semgrep rule can be used to identify schemas that specify one of the affected formats with the `integer` type, potentially resulting in improper validation.

```
rules:
  - id: schema_int_format_with_integer_type
    message: Found a schema property using the ‘integer’ type with a format of
‘int32’, ‘int64’, ‘uint32’, or ‘uint64’. This will not correctly validate integers
due to a bug in lisk-validator.
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-either:
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'uint32',
                }
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'uint64',
                }
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'int32',
                }
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'int64',
                }
```

Figure 52.4: Semgrep rule to identify schemas that are affected by this issue

Exploit Scenario

A method whose security assumptions depend on its parameters falling within the `int32/int64` or `uint32/uint64` ranges has a schema that renders this validation ineffectual. An attacker identifies this method and passes its values outside the required range, which breaks the method's assumptions and potentially allows attacks of higher severity, similar to the one described in TOB-LISK-24.

Recommendations

Short term, modify affected schemas to use the `string` type instead of `integer` to ensure that they are validated properly.

Long term, review `lisk-validator`'s handling of number types to ensure that integer properties provided as numbers are validated like their string counterparts.

91. totalWeight may exceed MAX_UINT64 in mainchain registration command verification

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-91
Target: `lisk-sdk/framework/src/modules/interoperability/sidechain/commands/register_mainchain.ts`	

Description

The `verify` method of the `MainchainRegistrationCommand` class iterates through the provided list of mainchain validators, summing their BFT weights into the `totalWeight` variable. If `totalWeight` exceeds the `MAX_UINT64` constant, the loop terminates, and the method returns an error (lines 132–137 in figure 91.1).

```
112    for (let i = 0; i < mainchainValidators.length; i += 1) {
113          const currentValidator = mainchainValidators[i];
114
115          if (
116                 mainchainValidators[i + 1] &&
117                 currentValidator.blsKey.compare(mainchainValidators[i +
1].blsKey) > -1
118          ) {
119                 return {
120                        status: VerifyStatus.FAIL,
121                        error: new Error('Validators blsKeys must be unique and
lexicographically ordered.'),
122                 };
123          }
124
125          if (currentValidator.bftWeight <= 0) {
126                 return {
127                        status: VerifyStatus.FAIL,
128                        error: new Error('Validator bft weight must be positive
integer.'),
129                 };
130          }
131
132          if (totalWeight > MAX_UINT64) {
133                 return {
134                        status: VerifyStatus.FAIL,
135                        error: new Error('Total BFT weight exceeds maximum
value.'),
136                 };
137          }
```

```
138
139         totalWeight += currentValidator.bftWeight;
140     }
```

Figure 91.1:
lisk-sdk/framework/src/modules/interoperability/sidechain/commands/register_mainchain.ts#112–140

However, this check is performed prior to the statement that adds the current weight to `totalWeight` in the loop (line 139 in figure 91.1). As a result, if the loop completes immediately after the addition causes `totalWeight` to exceed `MAX_UINT64`, the check will not occur. This leads to a violation of the assumption that `totalWeight` is less than or equal to `MAX_UINT64`.

Recommendations

Short term, move the check in lines 132–137 after the addition statement in line 139 of figure 91.1.

Long term, review all validation methods for similar issues. Add unit tests to check that validators correctly handle values that are near the limits (e.g., `MAX_UINT64`, `MAX_INT32`).

97. Extraneous feeTokenID option configured for token module

Severity: **Informational**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-LISK-97
Target: `lisk-core`	

Description

The available configuration options for Lisk SDK's `token` module are `userAccountInitializationFee` and `escrowAccountInitializationFee` (figure 97.1).

```
21    export interface ModuleConfig {
22          userAccountInitializationFee: bigint;
23          escrowAccountInitializationFee: bigint;
24    }
```

Figure 97.1: Configuration options for the `token` module
(`lisk-sdk/framework/src/modules/token/types.ts#21–24`)

However, Lisk Core's configuration files for all networks set the `feeTokenID` option within the `token` block, which is not a valid option for the module. For example, the `token` module's configuration in `lisk-core/config/mainnet/config.json` is given in figure 97.2:

```
87    token: {
88          feeTokenID: 0000000000000000
89    }
```

Figure 97.2: Mainnet `token` module configuration sets an unsupported option
(`lisk-core/config/mainnet/config.json#87–89`)

Exploit Scenario

A user starts Lisk Core with the provided mainnet configuration, which sets a value for `feeTokenID` in the block for the `token` module. Because this is not a valid option, the setting has no effect on the module's behavior.

Recommendations

Short term, remove `feeTokenID` from the `token` module configuration files in Lisk Core.

Long term, use the already defined schemas to perform validation within the module to ensure that its configuration does not include invalid options. With these checks in place, other misconfigurations may be detected as well.

99. LIP-0037 specifies ambiguous requirements for tags	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-99
Target: `lisk-sdk`, LIP-0037	

Description

LIP-0037 specifies message tags as strings with the `LSK_` prefix, an underscore [_] suffix, and content that is "a unique string indicating the scheme and, optionally, additional information." The content of a newly specified tag (specified outside the LIP) "must contain only ASCII characters between `0x21` and `0x7e` (inclusive), except that it must not contain underscore (`0x5f`)," according to the LIP.

The LIP specifies a set of message tags. One of these, `MESSAGE_TAG_CHAIN_REG_MESSAGE`, has the value `LSK_CHAIN_REGISTRATION_`. The tag violates the requirement of not having an underscore character except in the prefix and suffix. Although the requirement is specified only for new tags, it would be better to force all tags to adhere to it because if the purpose of the requirement is to ensure domain separation for hashing, then the tag in question may cause security issues in the future.

The issue is set only to informational because the issue has no practical consequences.

Exploit Scenario

A new tag format is added in a new release of Lisk. The new format's prefix is `LSK_CHAIN_`. Developers add a new tag in the new format. Domain separation for hashing is broken and attackers can swap signed messages of different types.

Recommendations

Short term, replace the `MESSAGE_TAG_CHAIN_REG_MESSAGE` tag with one that meets the requirements for new tags. Alternatively, remove the requirement from the LIP if it has no purpose.

Long term, ensure tag properties are validated in the code so it meets the assumptions that the creators of LIP-0037 implied.

100. BLS library does not properly check secret key

Severity: **Informational**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-LISK-100
Target: `lisk-sdk/elements/lisk-cryptography/src/bls_lib`	

Description

The `SecretKey.fromBytes` method performs some validation on the input, such as ensuring the secret key is nonzero and of the correct length. However, the check against the secret key is not done modulo the order of the elliptic curve. Therefore, it is possible for a secret key equivalent to zero to be used if the value of the secret key is a multiple of this elliptic curve order.

```
static fromBytes(skBytes: Uint8Array): SecretKey {
  if (skBytes.length !== SECRET_KEY_LENGTH) {
    throw new ErrorBLST(BLST_ERROR.BLST_INVALID_SIZE);
  }
  if (isZeroBytes(skBytes)) {
    throw new ErrorBLST(BLST_ERROR.ZERO_SECRET_KEY);
  }
  const sk = new SkConstructor();
  sk.from_bendian(skBytes);
  return new SecretKey(sk);
}
```

Figure 100.1: The `SecretKey.fromBytes` function with missing validation modulo the curve order (`blst-ts/src/lib.ts#63–73`)

The issue is unlikely to result in an exploit, but it could lead to unexpected or unintended behavior; it is expected that a mature library performs this validation correctly.

Exploit Scenario

A component of Lisk SDK uses the `blsSign` method, which calls the `SecretKey.fromBytes` method, to produce BLS signatures. Due to some other implementation mistake, a user uses a secret key value equivalent to the order of the elliptic curve; this value is passed to the `blsSign` function, resulting in the use of a zero key (whose corresponding public key is the point at infinity) to sign messages. A malicious user notices the use of the point at infinity as a public key and forges and impersonates messages for the other user.

Recommendations

Short term, add checks to all functions using the `SecretKey.fromBytes` function to ensure that the secret key being used for signing and other operations is nonzero modulo the order of the elliptic curve.

Long term, expand unit tests so that they check the correctness of various functionalities when a secret BLS key that is a multiple of the elliptic curve order is used. Review known issues in the cryptographic libraries used by `lisk-sdk` and either fix them in the wrapper methods inside the `lisk-sdk` codebase, track them internally, or document them. Ensure that all BLS public keys used in the system are verified with the `blsPopVerify` method since the `PublicKey.fromBytes` function does not validate the public keys it receives.

References

- BLS secret key validation is missing #96, ChainSafe/bls

Detailed Findings—Lisk DB

3. Invalid common prefix method	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-3
Target: `lisk-db/src/utils.rs`	

Description

The `common_prefix` function (figure 3.1) is incorrect. This function should return a vector with the common prefix of the two input vectors. Instead, it returns a vector of all the items that are equal in the two input vectors. The second `if` branch is missing an `else` counterpart with a `return` statement that would short circuit the `for` loop if one of the input vector's items does not match (i.e., we reach the end of the common prefix).

```
pub fn common_prefix(a: &[bool], b: &[bool]) -> Vec<bool> {
    let mut result = vec![];
    let (longer, shorter) = find_longer(a, b);
    for (i, v) in longer.iter().enumerate() {
        if i >= shorter.len() {
            return result;
        }
        if *v == shorter[i] {
            result.push(*v);
        }
    }
    result
}
```

Figure 3.1: The highlighted statement should have an `else` branch with a `return` statement. (`lisk-db/src/utils.rs#74–86`)

The `common_prefix` function is used in the `verify_query_keys` function, called by the `SparseMerkleTree::verify` function, as shown in figure 3.2. Therefore, the proof validation is invalid and can be bypassed.

```
let key_binary = utils::bytes_to_bools(key);
let query_key_binary = utils::bytes_to_bools(query.key());
let common_prefix = utils::common_prefix(&key_binary, &query_key_binary);
let binary_bitmap = utils::strip_left_false(&utils::bytes_to_bools(&query.bitmap));
```

```
if binary_bitmap.len() > common_prefix.len() {
    return false;
}
```

Figure 3.2: Use of the `common_prefix` function
(`lisk-db/src/sparse_merkle_tree/smt.rs#878–884`)

Exploit Scenario

An adversary exploits the bug in the sparse Merkle tree verification to provide fake proofs for cross-chain token transfers, effectively stealing tokens.

Recommendations

Short term, add the missing `return` statement in the `common_prefix` function. Figure 3.3 shows an example solution.

```
pub fn common_prefix(a: &[bool], b: &[bool]) -> Vec<bool> {
    let mut result = vec![];
    let (longer, shorter) = find_longer(a, b);
    for (i, v) in longer.iter().enumerate() {
        if i >= shorter.len() {
            return result;
        }
        if *v == shorter[i] {
            result.push(*v);
        } else {
            return result
        }
    }
    result
}
```

Figure 3.3: Example of code that fixes the `common_prefix` function

Long term, write more complex unit tests.

4. Panic due to lack of validation for Proof's bitmap length	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-4
Target: `lisk-db/src/sparse_merkle_tree/smt.rs`	

Description

The `query.bitmap` data—part of the `Proof` struct that is controlled by a user—is not correctly validated; there is no check limiting its length. If a proof with a long `query.bitmap` is provided, the `SparseMerkleTree::verify` function panics in the `prepare_queries_with_proof_map` method with an out-of-range error. The error triggers in the array slicing operation, highlighted in figure 4.1.

```
pub fn prepare_queries_with_proof_map(
    proof: &Proof,
) -> HashMap<Vec<bool>, QueryProofWithProof> {
    let mut queries_with_proof: HashMap<Vec<bool>, QueryProofWithProof> =
HashMap::new();
    for query in &proof.queries {
        let binary_bitmap =
utils::strip_left_false(&utils::bytes_to_bools(&query.bitmap));
        let binary_path =
utils::bytes_to_bools(&query.pair.0)[..binary_bitmap.len()].to_vec();
```

Figure 4.1: Sub-function of the `SparseMerkleTree::verify` function where a panic may occur (`lisk-db/src/sparse_merkle_tree/smt.rs#1363–1370`)

Figure 4.2 shows a unit test that demonstrates the error.

```
#[test]
fn test_panic() {
    let mut data = UpdateData::new_from(Cache::new());
    let keys =
vec!["bbbbc758f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879"];
    let values =
vec!["9c12cfdc04c74584d787ac3d23772132c18524bc7ab28dec4219b8fc5b425f70"];
    for i in 0..keys.len() {
        data.insert(SharedKVPair(&hex::decode(keys[i]).unwrap(),
&hex::decode(values[i]).unwrap()));
    }

    let mut tree = SparseMerkleTree::new(&[], KeyLength(32), Default::default());
    let mut db = smt_db::InMemorySmtDB::default();
    let root = tree.commit(&mut db, &data).unwrap();
```

```
    let mut proof = tree.prove(&mut db, &keys.iter().map(|k|
hex::decode(k).unwrap()).collect::<NestedVec>()).unwrap();

    proof.queries[0].bitmap =
Arc::new(vec![1u8,2,3,4,1,3,3,71,3,3,71,3,3,71,3,3,71,3,3,71,3,3,71,3,3,71,3,3,71,3,
3,71,3,3,7]);

    let is_valid = SparseMerkleTree::verify(&keys.iter().map(|k|
hex::decode(k).unwrap()).collect::<NestedVec>(),
        &proof, &root.lock().unwrap(), KeyLength(32)).unwrap();
    assert!(is_valid);
}
```

Figure 4.2: Unit test triggering a panic in the `SparseMerkleTree::verify` function

The effect of triggering this error in the TypeScript code is an infinite hang, which waits for the Rust process to return data even though the process is already terminated.

Exploit Scenario

An attacker sends a maliciously constructed proof to a validator node. The node hangs, stops processing data, and is slashed.

Recommendations

Short term, add validation of `query.bitmap`'s length in the `SparseMerkleTree::verify` function.

Long term, create fuzz tests to try to find panics in `SparseMerkleTree` class methods.

5. Sparse Merkle tree proof's verification algorithm is invalid

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-LISK-5
Target: `lisk-db/src/sparse_merkle_tree/smt.rs`	

Description

The proof verification function in `lisk-db` contains multiple algorithm-level bugs. The algorithm from the original specification (shown in figure 5.1) takes as inputs a key-value pair whose existence should be proved, a proof structure with an optional leaf node (that contains some key-value pair) and an array of siblings, and the expected root digest. The function handles both existence (inclusion) and nonexistence (noninclusion) proofs.

```
Algorithm 1: JMT Proof Verification
Input  : Node key k, proof p, the expected root digest d_root, and the node value v that needs to
         be verified against.
Output: return true if the v is verified by p and vice versa

if v ≠ NULL then                                        // Node existence expects inclusion proof
    if p.leaf ≠ NULL then
        // Prove inclusion with inclusion proof
        if k ≠ p.leaf.key or hash(blob) ≠ p.leaf.value_hash then
            return false
        end
    else            // Expected inclusion proof but get non-inclusion proof passed in
        return false
    end
else                                                    // Node absense expects exclusion proof
    if p.leaf ≠ NULL then                               // The inclusion proof of another node
        if k = p.leaf.key or CommonPrefixLengthInBits(p.leaf.key) < Len(p.siblings) then
            return false
        end
    else                                                // Prove exclusion with an exclusion proof
        // Noop
    end
end
if p.leaf = NULL then d_cur ← D_default else d_cur ← hash(p.leaf)
for i ← DigestLengthInBits − Len(p.siblings) − 1 to 0 do
    if the i^th bit from MSB of k = 1 then
        d_cur ← hash(p.siblings[i] ‖ d_cur)
    else
        d_cur ← hash(d_cur ‖ p.siblings[i])
    end
end
return d_cur = d_root
```

Figure 5.1: Original Jellyfish proof verification algorithm (Jellyfish Merkle Tree white paper)

Value	Leaf value	Proof type	Action
NOT NULL	NOT NULL	Existence	Assert key == leaf key
NOT NULL	NULL	Existence	Fail verification
NULL	NOT NULL	Nonexistence	Assert key != leaf key Assert length of common prefix >= length of siblings
NULL	NULL	Nonexistence	Do nothing

Table 5.1: Explanation of the original Jellyfish proof verification algorithm: the first two columns specify inputs, the third column specifies the proof type implied by the inputs, and the last column specifies validations performed by the algorithm.

Existence proofs are straightforward: The prover must include the requested key and value in the proof.

For nonexistence proofs, there are three cases. The provided path (constructed from the key and siblings) may lead to the following:

A. a subtree with exactly one leaf with key K2 and some value, where K2 does not equal the requested key

B. an empty subtree

C. a leaf with a key equal to the requested key and value V2, where V2 does not equal the requested value

Case C is not handled by the Diem algorithm.

The corresponding verification function in Lisk's code also handles both cases in a single method. However, Lisk's function does not take a key–value pair as input; it takes only the key. This limits the number of cases that the function must handle because the value and the leaf value are the same.

Value	Leaf value	Proof type	Action
NOT NULL	NOT NULL	Existence	If key != leaf key then assert length of common prefix >= length of siblings
NULL	NULL	Nonexistence	

Table 5.2: explanation of the Lisk's proof verification algorithm

```
if utils::is_bytes_equal(key, query.key()) {
    continue;
}
```

```
let key_binary = utils::bytes_to_bools(key);
let query_key_binary = utils::bytes_to_bools(query.key());
let common_prefix = utils::common_prefix(&key_binary, &query_key_binary);
let binary_bitmap = utils::strip_left_false(&utils::bytes_to_bools(&query.bitmap));
if binary_bitmap.len() > common_prefix.len() {
    return false;
}
```

Figure 5.2: The code corresponding to lines 1–16 in the white paper (`lisk-db/src/sparse_merkle_tree/smt.rs#875–884`)

Because of how the algorithm is implemented, existence verification can be bypassed by providing keys that share a common prefix with keys in the tree and have the same values (see figure 5.2).

```
#[test]
fn test_inclusion_vuln() {
    let mut data = UpdateData::new_from(Cache::new());
    let keys =
vec!["aaaac758f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879",
"bbbbc758f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879"];
    let keys2 =
vec!["accc0000f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879",
"bddd1111f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879"];
    let keys3 =
vec!["00000000f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879",
"00001111f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879"];

    let values =
vec!["eeeec758f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879",
"ffffc758f6d27e6cf45272937977a748fd88391db679ceda7dc7bf1f005ee879"];

    for i in 0..keys.len() {
        data.insert(SharedKVPair(&hex::decode(keys[i]).unwrap(),
&hex::decode(values[i]).unwrap()));
    }
    let mut tree = SparseMerkleTree::new(&[], KeyLength(32), Default::default());
    let mut db = smt_db::InMemorySmtDB::default();
    let root = tree.commit(&mut db, &data).unwrap();

    let mut proof = tree.prove(&mut db, &keys.iter().map(|k|
hex::decode(k).unwrap()).collect::<NestedVec>()).unwrap();

    let is_valid = SparseMerkleTree::verify(&keys2.iter().map(|k|
hex::decode(k).unwrap()).collect::<NestedVec>(), &proof, &root.lock().unwrap(),
KeyLength(32)).unwrap();
    assert!(is_valid); // verification succeeded
```

```
    let is_valid2 = SparseMerkleTree::verify(&keys3.iter().map(|k|
hex::decode(k).unwrap()).collect::<NestedVec>(), &proof, &root.lock().unwrap(),
KeyLength(32)).unwrap();
    assert!(!is_valid2); // verification failed
}
```

Figure 5.3: Unit test demonstrating that inclusion (existence) verification is invalid

Moreover, nonexistence verification is undefined:

A. There is a correct assertion (implicit via the `if` statement) that the leaf key does not equal the requested key and that the common prefix is long enough. However, there is no check to ensure that the value is not null. Therefore, this case is handled incorrectly. Exploitability of this issue was not determined, but in the worst case, it may allow bypassing nonexistence verification.

B. There is a redundant validation of a leaf key, which may make it impossible to construct valid nonexistence proofs in some cases.

C. This case is not handled, as there is no distinction between the requested value (whose nonexistence we want to verify) and the value in a proof.

Furthermore, the Lisk algorithm allows proving inclusion or noninclusion of multiple keys at once, which could lead to more potential confusion if some of the keys are proved to exist and some others are proved to not exist.

Exploit Scenario

An attacker exploits the bug in figure 5.3 in the sparse Merkle tree verification to construct fake proofs for cross-chain token transfers. He uses the proofs to steal tokens.

Recommendations

Short term, review the sparse Merkle tree algorithm design. Fix handling of the various cases described in the finding. Make the `lisk-db` API easy to consume.

Long term, create differential fuzzing tests against other libraries implementing sparse Merkle trees.

7. Lack of length validation for leaf keys in sparse Merkle tree proof verification

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-7
Target: `lisk-db/src/sparse_merkle_tree/smt.rs`	

Description

The `SparseMerkleTree::verify` function does not check lengths of leaf keys in the provided proof.

Not checking lengths of keys may allow an attacker to perform a confusion attack for the key-value pairs. For example, if the expected key length is X bytes, the attacker may send a key of length X + 2 and shorten the value by two bytes while keeping the leaf hash unchanged. Figure 7.1 presents the leaf hashing function used in the proof for the `SparseMerkleTree::verify` function.

```
impl Hash256 for KVPair {
    fn hash(&self) -> Vec<u8> {
        let mut hasher = Sha256::new();
        hasher.update(PREFIX_LEAF_HASH);
        hasher.update(self.key());
        hasher.update(self.value());
        let result = hasher.finalize();
        result.to_vec()
    }
}
```

Figure 7.1: Function hashing keys and values that are part of a proof (`lisk-db/src/sparse_merkle_tree/smt.rs#173–182`)

It should be possible to fix key lengths to a constant value, and therefore enable length validation, by hashing the keys. Hashing keys is mentioned in LIP-0039 as a recommendation.

> As mentioned, the tree height is O(Log(N)) if the keys are randomly distributed. It is easy to fulfill this condition by hashing the keys prior to insertion in the tree. The average number of operations needed to update the tree after inserting a node approaches Log(N), while it equals L (256 for 32-byte keys) in the worst case (two consecutive leaf nodes differing only on their last key digit). The additional introduction of extension nodes could eliminate empty nodes and therefore bring the number of operations down to O(Log(N)) also for the worst case. However, this optimization introduces extra complexity in the non-inclusion proof protocol, and as explained, this drawback is not relevant for randomly distributed keys.

Figure 7.2: Part of LIP-0039 mentioning that hashing keys is recommended

The difficulty of the finding is set to high because for all reviewed uses of the sparse Merkle tree, the values being hashed in a vulnerable method are of constant size (and are themselves hashes of actual values). Therefore, exploiting the vulnerability should not be possible.

Exploit Scenario
The developers of Lisk release a new version of the product with code that uses the sparse Merkle tree verification, where the values included in the tree are not hashed before inclusion and the keys are attacker controlled. An attacker exploits the lack of length validation to create fake proofs.

Recommendations
Short term, modify the code to either add length validation of keys provided in a proof, hash the keys before using them, or introduce domain separation for leaf hashing.

8. Lack of sparse Merkle tree personalized tree-wide constant	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-8
Target: `lisk-db`	

Description

To prevent attacks across distinct trees (trees used by systems other than Lisk), a tree-wide constant should be introduced. A tree-wide constant is unique to Lisk and not used by any other sparse Merkle tree (SMT) in the world and is prepended to every leaf node's data. Such a constant would increase Lisk's tree security in a multi-instance setting where an attacker finds hash collisions for some tree (e.g., a tree used in Ethereum) and applies the collisions directly to Lisk's tree.

Introducing a unique tree-wide constant decreases the risk resulting from the probably impossible event that someone finds a SHA-256 collision. If such an event happens, an attacker will have to separately do the same collision-finding work to exploit the Lisk tree's implementation.

Recommendations

Short term, introduce a unique tree-wide constant to the `lisk-db` SMT implementation. The constant should be used as part of node encoding, either in every call to a hash function or only for leaf node encoding.

Long term, research what constitutes a secure node encoding for Lisk's SMTs. For example, consult section 5.2 from the white paper listed in the References section.

References

- Rasmus Dahlberg, Tobias Pulls, and Roel Peeters, "Efficient Sparse Merkle Trees," Nordic Conference on Secure IT Systems, 2016.

Detailed Findings—Lisk Desktop

53. Electron version is outdated and uses vulnerable Chromium version	
Severity: **Medium**	Difficulty: **High**
Type: Patching	Finding ID: TOB-LISK-53
Target: `lisk-desktop`	

Description

The `lisk-desktop` application uses Electron version 17.2.0, which reached end of life in August 2022, as shown in Electron's version Timeline. At the time of this writing, the latest Electron version is 25.0.0.

By running an old Electron version, `lisk-desktop` is also running an old Chromium version. The latest version of Chromium is M114, but `lisk-desktop` uses version M98, a version released in February 2022 that contains several known one-day vulnerabilities.

The `lisk-desktop` application also contains multiple other vulnerable dependencies. Running `npm audit` yields 20 high-, 109 medium- and 3 low-severity vulnerabilities. We did not assess whether these vulnerabilities impact `lisk-desktop`.

Exploit Scenario

An attacker finds a cross-site scripting (XSS) vulnerability in `lisk-desktop` that is reachable through the `lisk://` protocol. The attacker leverages the XSS and a one-day vulnerability (e.g., CVE-2023-2033) to exploit the vulnerable Chromium instance and obtain remote code execution on the user's machine.

Recommendations

Short term, update `lisk-desktop` to use the latest version of Electron. Update every other vulnerable dependency where possible. Run `npm audit` to confirm that no vulnerable dependencies remain in `lisk-desktop`.

Long term, create a process to update the Electron version soon after a new version is released. Implement dependency checks (e.g., `npm audit`) as part of the CI/CD pipeline. Do not allow builds to continue with any outdated dependencies.

References

- Security: Use a current version of Electron, Electron
- s1r1us, "Discord Desktop – Remote Code Execution," *Electrovolt*, 2022.

54. Electron renderer lacks sandboxing	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-54
Target: `lisk-desktop`	

Description

The `lisk-desktop` application creates its main `BrowserWindow` instance (where the application will render) without the `sandbox` option, which enables the renderer's sandbox. This option became the default in version 20 of Electron; however, `lisk-desktop` currently uses version 17.

```
win.browser = new BrowserWindow({
  width: width > 1680 ? 1680 : width,
  height: height > 1050 ? 1050 : height,
  minHeight: 576,
  minWidth: 769,
  center: true,
  webPreferences: {
    // Avoid app throttling when Electron is in background
    backgroundThrottling: false,
    // Specifies a script that will be loaded before other scripts run in the page.
    preload: path.resolve(__dirname, '../src/ipc.js'),
    //
https://www.Electronjs.org/docs/latest/tutorial/security#isolation-for-untrusted-con
tent
    nodeIntegration: false,
    contextIsolation: true,
  },
});
```

Figure 54.1: `BrowserWindow` creation without the sandbox option enabled
`lisk-desktop/app/src/modules/win.js#12–27`

Exploit Scenario

An attacker finds a cross-site scripting (XSS) vulnerability in `lisk-desktop` that is reachable through the `lisk://` protocol. The attacker leverages the XSS and a one-day vulnerability (e.g., CVE-2023-2033) to exploit the vulnerable Chromium instance and obtain remote code execution on the user's machine. Since there is no sandbox, the attacker does not need to use a sandbox exploit to obtain full remote code execution.

Recommendations

Short term, enable sandboxing on the application's `BrowserWindow` by having the code pass the `sandbox: true` option during the `BrowserWindow` initialization. Alternatively, updating Electron to a version later than 20 will also enable sandboxing by default.

References

- Security: Enable process sandboxing, Electron
- Process Sandboxing, Electron

55. Lack of a CSP in lisk-desktop

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-55
Target: `lisk-desktop`	

Description

The `lisk-desktop` application renders its content without a Content Security Policy (CSP). A CSP adds extra protection against cross-site scripting (XSS) and data injection by allowing developers to specify which source the browser can execute or render code from.

Opening the DevTools tab in the application shows the warning about the lack of a CSP (figure 55.1).

```
⚠ ▸Electron Security Warning (Insecure Content-Security-Policy) This renderer process
has either no Content Security
  Policy set or a policy with "unsafe-eval" enabled. This exposes users of
  this app to unnecessary security risks.
```

Figure 55.1: A warning message about the lack of a CSP in the application's DevTools

Exploit Scenario

An attacker finds an XSS vulnerability in `lisk-desktop` that is reachable through the `lisk://` protocol. The attacker has no trouble injecting JavaScript since there is no CSP to prevent inline execution of JavaScript.

Recommendations

Short term, define a CSP in the application's `index.html` page using the `meta` tag:

```
<meta http-equiv="Content-Security-Policy" content="default-src
'none'">
```

Validate the CSP with a CSP Evaluator.

References

- Content Security Policy (CSP), Mozilla
- Security: Define a Content-Security-Policy, Electron
- oskarsv, "Remote Code Execution in Slack desktop apps + bonus," HackerOne, January 27, 2020.

56. Electron app does not validate URLs on new windows and navigation

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-56
Target: `lisk-desktop`	

Description

The `will-navigate` and `new-window` handlers—triggered on redirects and window creation events from the main window—flow the new URL into the `electron.shell.openExternal` function without any further validation, as shown in figure 56.1.

```
const handleRedirect = (e, url) => {
  if (url !== win.browser.webContents.getURL()) {
    e.preventDefault();
    electron.shell.openExternal(url);
  }
};
win.browser.webContents.on('will-navigate', handleRedirect);
win.browser.webContents.on('new-window', handleRedirect);
```

Figure 56.1: `lisk-desktop/app/src/modules/win.js#85–92`

The `openExternal` function is known to lead to remote code execution (RCE) if the URL is controlled by an attacker, as explained in the blog post The dangers of Electron's `shell.openExternal()`—many paths to remote code execution and demonstrated in the exploits of Rocket.Chat and WordPress Desktop (among others).

Exploit Scenario

An attacker finds a way to inject an arbitrary link in the website (e.g., through a malicious app) and injects a link with the `smb://` protocol. The user clicks the malicious link, which downloads and runs the malicious binary. The attacker gains RCE on the user's machine.

Recommendations

Short term, have the code validate that the URLs flowing into the `openExternal` function use the HTTP or HTTPS protocol. If only a subset of domains is expected (e.g., only redirects to `lisk.com`), have the code validate them as well. Use the JavaScript URL object, instead of string comparisons, to do these checks.

References

- Security: Do not use `shell.openExternal` with untrusted content, Electron

57. IPC exposes overly sensitive functionality

Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-LISK-57
Target: `lisk-desktop`	

Description

The Electron preload script exposes the `send`, `on`, `once`, and `removeListener` functions of the `ipcRenderer` object to the renderer application (the React app), as shown in figure 57.1.

```
contextBridge.exposeInMainWorld('ipc', {
  send: (channel, data) => {
    ipcRenderer.send(channel, data);
  },
  on: (channel, func) => {
    ipcRenderer.on(channel, (event, ...args) => {
      func(event, ...args);
    });
  },
  once: (channel, func) => {
    ipcRenderer.once(channel, (event, ...args) => {
      func(event, ...args);
    });
  },
  removeListener: (channel, func) => {
    ipcRenderer.removeListener(channel, (event, ...args) => {
      func(event, ...args);
    });
  },
});
```

Figure 57.1: `lisk-desktop/app/src/ipc.js#5–24`

This is not recommended because it may expose much more functionality to the renderer than necessary. Electron's documentation makes this very clear, as shown in figure 57.2.

> **SECURITY WARNING**
>
> We don't directly expose the whole `ipcRenderer.send` API for **security reasons**.
>
> Make sure to limit the renderer's access to Electron APIs as much as possible.

Figure 57.2: `https://www.electronjs.org/docs/latest/tutorial/ipc`

Exploit Scenario

An attacker finds a cross-site scripting (XSS) vulnerability in `lisk-desktop` that is reachable through the `lisk://` protocol. The attacker uses the overly permissive IPC permissions to escalate his privileges from the renderer process to the main process.

Recommendations

Short term, expose each functionality (message that the renderer can send or receive) in a specific function with a clear purpose. Follow Electron's IPC tutorial to learn how to effectively send messages from the main process to the renderer and vice versa. This will limit the renderer's access to Electron's APIs to the minimum.

References

- Inter-Process Communication, Electron

58. Electron local server is exposed on all interfaces	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-58
Target: `lisk-desktop`	

Description

The `lisk-desktop` application launches a local web server on port `5659` that is exposed on all interfaces (figure 58.1).

```
tcp6       0      0 :::5659                 :::*                    LISTEN
```

Figure 58.1: The result of the `netstat -tulnp` command, showing that `lisk-desktop` exposes a local web server on all interfaces

Figure 58.2 shows the code that creates the server. By not passing the `host` argument to the `listen` function, the express server will listen on all interfaces.

```
const server = {
  init: (port) => {
    // [REDACTED]
    const app = express();
    // [REDACTED]
    app.listen(port);
    // [REDACTED]
  },
};
```

Figure 58.2: `lisk-desktop/app/server.js#4–26`

This allows other devices on the same local network to make requests to the server (e.g., with a local IP address such as `192.168.X.X`).

Exploit Scenario

An attacker finds a vulnerability in the local server that enables him to gain privileges on the local machine (e.g., local file read, local file write, remote code execution). A user has the Lisk application running in the background while connected to the public Wi-Fi of a blockchain conference. The attacker, who is also connected to the conference Wi-Fi, connects to the target user's server, exploits the vulnerability, and elevates his privileges on the target user's machine, potentially stealing their funds.

Recommendations

Short term, modify the call to the `listen` function to include the `host` argument, such as `app.listen(port, "127.0.0.1")`. This will ensure that attackers on the same network as the user cannot access the user's local server.

59. Unnecessary use of innerHTML

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-59
Target: `lisk-desktop`	

Description

The code unnecessarily uses the `.innerHTML` function to append simple text to the DOM, as shown in figure 59.1. This function allows users to modify the HTML of an element; however, in this case, this is unnecessary because the `data-name` attribute of icon elements is always simple text without HTML tags. Using the `.innerText` attribute is sufficient.

```
<script>
  (function () {
    document.getElementById('icons').onclick = function (e) {
      e = e || window.event;
      var name =
        e.target.getAttribute('data-name') ||
e.target.parentNode.getAttribute('data-name');
      document.getElementById('name').innerHTML = name;
    };
  })();
</script>
```

Figure 59.1: *`lisk-desktop/setup/react/assets/fonts/iconfont/icons.html#497–506`*

This finding is set to informational severity because attacker input cannot reach the `.innerHTML` sink.

Exploit Scenario

The source code is modified so that an attacker can add links with a controlled `data-name` attribute. The attacker injects malicious JavaScript code and steals a user's funds.

Recommendations

Short term, modify the code to use `.innerText` instead of `.innerHTML`. This will accomplish the same result without the risk of cross-site scripting (XSS).

Long term, review every other HTML sink and, where really necessary, add a comment explaining why it exists, and why unsanitized attacker input cannot reach it.

References

- HTMLElement: innerText property, Mozilla
- Element innerHTML property, Mozilla

60. ReDoS in isValidRemote function	
Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-LISK-60
Target: `lisk-desktop`	

Description

The `isValidRemote` function contains the regex shown in figure 60.1, which is vulnerable to a regular expression denial-of-service (ReDoS) attack. A regex that is vulnerable to ReDoS attacks tests inputs exponentially more slowly in relation to the input length, which exhausts a server's resources.

```
const isValidRemote = (url) =>
  /(([a-z\d]([a-z\d-]*[a-z\d])*)\.)+[a-z]{2,}|((\d{1,3}\.){3})/.test(url.hostname);
```

Figure 60.1: Regex vulnerable to ReDoS (`lisk-desktop/src/utils/login.js#12–13`)

Passing an input with several repeated `0`'s to this regex test causes the application to hang indefinitely, as shown in figure 60.2.

```
/(([a-z\d]([a-z\d-]*[a-z\d])*)\.)+[a-z]{2,}|((\d{1,3}\.){3})/.test('0000000000000000
000000000000000')
```

Figure 60.2: Example of an input that causes the call to `test` to hang indefinitely

This finding is set to informational severity because the `isValidRemote` function is not used and therefore cannot be reached by an attacker.

Recommendations

Short term, modify the regex to prevent the ReDoS attack.

Long term, ensure that the `js/redos` CodeQL rule is included in the project's CI/CD pipeline, and prevent commits that have findings related to this rule from being pushed to the main branch. This will minimize the risk that other ReDoS-vulnerable regexes are deployed.

References

- ReDoS, Wikipedia

61. Improper handling of the custom lisk:// protocol in lisk-desktop	
Severity: **Undetermined**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-61
Target: `lisk-desktop`	

Description

The `lisk://` protocol handler does not sufficiently validate incoming data and contains a bug that prevents some functionality from working properly.

As highlighted in figure 61.1, the custom protocol handler on the main process sends a message to the `openUrl` handler of the renderer process (the React app).

```
const handleProtocol = () => {
  // Protocol handler for MacOS
  app.on('open-url', (event, url) => {
    event.preventDefault();
    win.browser?.show();
    win.send({ event: 'openUrl', value: url });
  });
};
```

Figure 61.1: The `lisk://` protocol handler (`lisk-desktop/app/src/main.js#60–67`)

The renderer process minimally processes the URL and uses it to redirect to another path of the application.

```
const sendRegex = /^\/(wallet|wallet\/send|main\/transactions\/send)$/;
const sendRedirect = '/wallet?modal=send';

const stakeRegex = /^\/(main\/staking\/stake|validator\/stake|stake)$/;
const stakeRedirect = '/wallet?modal=StakingQueue';

export const externalLinks = {
  init: () => {
    const { ipc } = window;

    if (ipc) {
      // eslint-disable-next-line max-statements
      ipc.on('openUrl', (_, url) => {
        const urlDetails = new URL(url);
        const { protocol, href, search } = urlDetails;

        // Due to some bug with URL().pathname displaying a blank string
```

```
        // instead of the correct pathname, it was best to use href with a regex
        const normalizedUrl = href.match(/\/\w+/)[0];
        const searchParams = search.slice(1);
        if (protocol?.slice(0, -1).toLowerCase() === 'lisk' && normalizedUrl) {
          let redirectUrl = normalizedUrl;
          if (normalizedUrl.match(sendRegex)) {
            redirectUrl = sendRedirect + (searchParams ? `&${searchParams}` : '');
          } else if (normalizedUrl.match(stakeRegex)) {
            redirectUrl = stakeRedirect + (searchParams ? `&${searchParams}` : '');
          }

          // @todo do we need to both push and replace?
          history.push(redirectUrl);
          history.replace(redirectUrl);
        }
      });
    }
  },
};
```

Figure 61.2: The `openUrl` handler, which is called from the `lisk://` protocol handler (`lisk-desktop/src/utils/externalLinks.js#3–38`)

The way the path name is parsed with a regex is incorrect (highlighted in red in figure 61.2) and only recovers part of a path. For example, `normalizedUrl` will be `/a` for `/a`, `/a/b`, and `/a-b`, which is incorrect. This means that a link to `main/staking/stake` will not redirect the user to `/wallet?modal=StakingQueue` as intended because the `normalizedUrl.match(stakeRegex)` check will not pass.

The finding is set to undetermined severity because more research needs to be done to determine whether attacker input can cause an arbitrary redirect and whether this can lead to a full compromise.

Exploit Scenario

An attacker finds a route that triggers an action without user confirmation. The attacker uses the `lisk://` protocol to send the user to this route, triggering the unwanted action.

Recommendations

Short term, have the code parse the incoming URL with the URL object. It is unlikely that this object will return incorrect results, as the comments imply. Create an allowlist of paths that the user can redirect to, and identify which search parameters are allowed for each (otherwise attackers may be able to use parameters such as `referrer` to bypass the allowlist). This will prevent an attacker from redirecting the user to any path of the application.

Long term, write negative tests to ensure that only the intended functionality is reachable from the custom `lisk://` protocol.

62. Lack of permission checks in the Electron application

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-62
Target: `lisk-desktop`	

Description

The code does not use the `setPermissionRequestHandler` function to prevent the renderer from accessing systems such as the webcam and notification systems, as specified in Electron's documentation:

> *By default, Electron will automatically approve all permission requests unless the developer has manually configured a custom handler. While a solid default, security-conscious developers might want to assume the very opposite.*

A browser such as Chrome does the opposite and asks the user for permission. In `lisk-desktop`, this is not the case, which may allow an attacker who can inject JavaScript into the application to read the user's clipboard or silently record audio and video.

Exploit Scenario

An attacker finds a cross-site scripting (XSS) vulnerability in `lisk-desktop` that is reachable through the `lisk://` protocol. The attacker reads the user's clipboard and silently records the user's audio and video.

Recommendations

Short term, implement `setPermissionRequestHandler` to prevent the renderer from having access to every permission. In the handler, allow only permissions that the application actually needs.

Long term, review Electron security best practices and ensure they are being used.

References

- Security: Handle session permission requests from remote content, Electron

73. Unnecessary XSS risk in htmlStringToReact	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-73
Target: `lisk-desktop`	

Description

The `htmlStringToReact` function uses React's `createElement` method to convert the application's release notes into React. The `createElement` function may lead to cross-site scripting (XSS) if an attacker can control some of its parameters. Example payloads include the following:

- ```
  createElement('div',{ dangerouslySetInnerHTML: {__html: "<h1>asd</h1>"} },);)
  ```
- ```
  createElement('iframe',{ srcdoc: "<h1>asd</h1><script>window.alert(1)</script>" },);
  ```

Instead of relying on custom logic and regexes to parse the release notes, the code should use a purposely built function to sanitize the HTML to mitigate XSS risks.

Exploit Scenario

An attacker finds a way to inject data into the release notes by contributing to the `lisk-desktop` project. The attacker finds a bypass on the `htmlStringToReact` function that allows him to control arbitrary props. He then uses one of the payloads above to cause XSS on every user that sees the release notes.

Recommendations

Short term, use a library such as `DOMPurify` to sanitize the HTML before processing it. This will minimize the risk of XSS.

80. WalletConnect integration crashes on requiredNamespaces without Lisk	
Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-80
Target: `lisk-desktop` WalletConnect integration	

Description

When a decentralized application (dapp) connects to the `lisk-desktop` wallet using WalletConnect with the `requiredNamespaces` parameter missing the `lisk` key, the desktop wallet hangs in the loading screen until a user closes and reopens it.

This issue can be triggered by connecting to the desktop wallet using this WalletConnect dapp example.

Exploit Scenario

A user tries to connect to a dapp that does not support Lisk. The `lisk-desktop` application hangs, hindering the user's experience.

Recommendations

Short term, have the code validate that the WalletConnect connection request's `requiredNamespace` parameter includes the `lisk` key. If the `lisk` namespace is not present, have the application show an error message to the user saying that the dapp is not supported.

81. WalletConnect integration accepts any namespaces	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-81
Target: `lisk-desktop` WalletConnect integration	

Description

The `lisk-desktop` wallet supports only the `lisk` namespace; however, it does not reject WalletConnect connections that list other namespaces in the `requiredNamespaces` parameter. As specified in WalletConnect's documentation, wallets should reject connections from decentralized applications (dapps) that request namespaces that the wallet does not support.

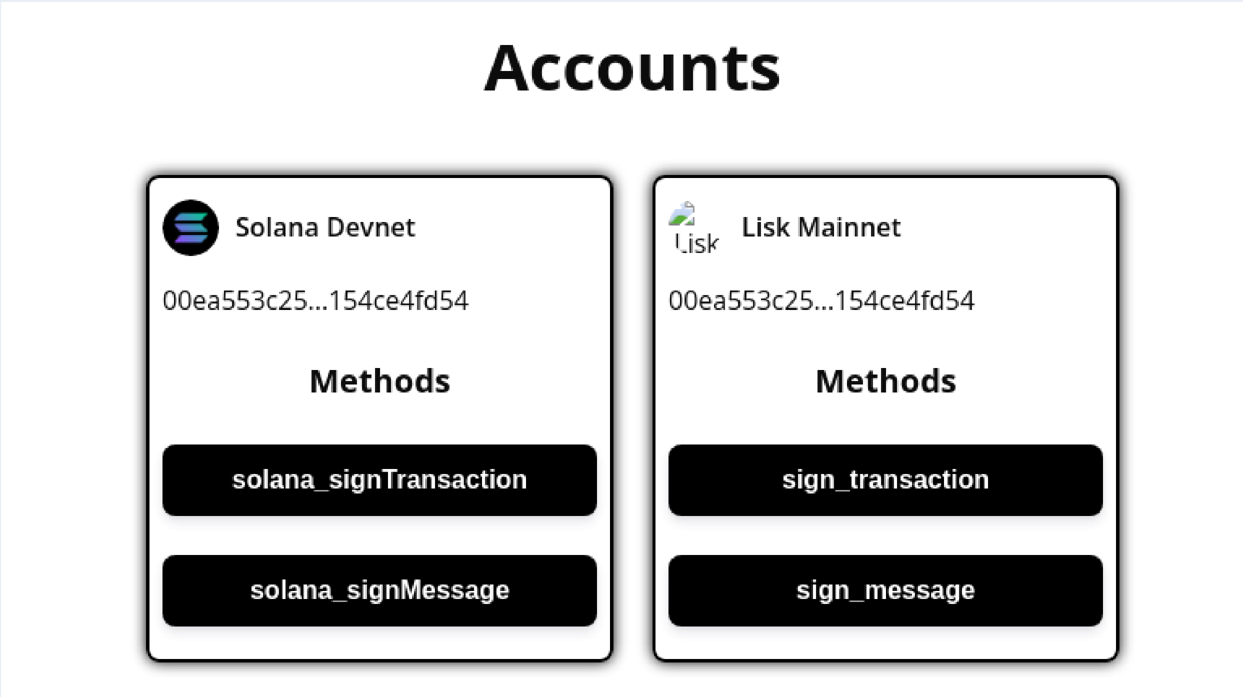

Figure 81.1: The `ManuGowda/walletConnect` app after connecting to the `lisk-desktop` wallet with both Lisk and Solana

Trying to sign transactions or messages with the Solana methods returns an error.

Recommendations

Short term, have the code reject any WalletConnect connection request that requires any namespace other than `lisk`.

82. Impossible to cancel a WalletConnect approval request without refreshing

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-82
Target: `lisk-desktop` WalletConnect integration	

Description

When a user rejects a WalletConnect connection, they cannot add another one until they close and reopen the desktop application because the old connection event is never deleted. This happens because the `reject` function does not call the `removeEvent(proposalEvents)` function like the `approve` function does.

```
const approve = useCallback(async (selectedAccounts) => {
  const proposalEvents = events.find((e) => e.name === EVENTS.SESSION_PROPOSAL);
  try {
    const status = await onApprove(proposalEvents.meta, selectedAccounts);
    removeEvent(proposalEvents);
    setSessionProposal(null);
    setSessionRequest(null);

    // [REDACTED]
}, []);

const reject = useCallback(async () => {
  const proposalEvents = events.find((e) => e.name === EVENTS.SESSION_PROPOSAL);
  try {
    await onReject(proposalEvents.meta);
    setSessionProposal(null);
    setSessionRequest(null);
    // [REDACTED]
}, []);
```

Figure 82.1: Image showing the lack of `removeEvent` in the `reject` function (`lisk-desktop/libs/wcm/hooks/useSession.js#37–73`)

Exploit Scenario

A user rejects a WalletConnect connection request and then clicks the “+ Connect Wallet” button to add a different decentralized application (dapp). The old, rejected request is shown again. The user becomes confused and adds the dapp they previously rejected.

Recommendations

Short term, have the code clear the event from the list of events on both approval and rejection. This will ensure that a processed event is not shown to the user again.

83. Desktop and mobile applications do not validate data coming from online services	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-83
Target: `lisk-desktop`, `lisk-mobile`	

Description

Both `lisk-desktop` and `lisk-mobile` receive data from online services, primarily instances of `lisk-service` maintained by the Lisk team. That data is used in various functionalities but most importantly in transaction construction and signing procedures. However, some information received from the online services is not sufficiently validated.

A proper validation must comprise two phases:

- Technical, in-code, invisible-to-a-user validation of syntactic and basic semantic properties: this type of validation includes, for example, validation of length, format, and correspondence to other data.
- Manual validation of the data by a user: users should be able to manually check and confirm data received from external (and so potentially malicious) endpoints.

Example instances of the lack of validation are listed below. More issues of this kind may exist in vulnerable codebases. However, due to time constraints, we did not investigate all possible attack vectors.

The first example is for the `lisk-desktop` application. When an approved decentralized application (dapp) triggers a transaction through WalletConnect, the user is prompted for approval in their desktop wallet. This approval process has two screens: the first one showing the dapp name, the dapp URL, and the chain ID (figure 83.1); and a second one showing a summary of the transaction (figure 83.2).

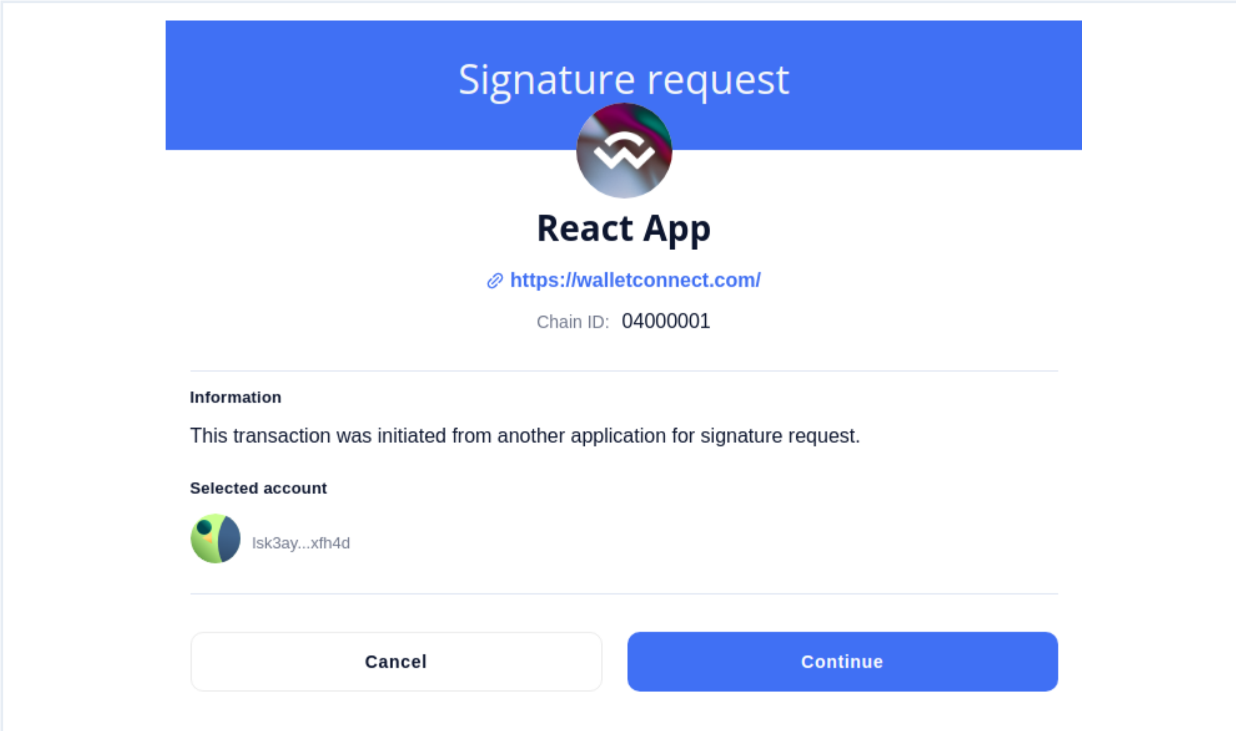

Figure 83.1: First screen of the Signature Request process

These screens should contain every piece of information the user needs to make an informed decision about whether to approve or reject the transaction. Specifically, the transaction summary screen (figure 83.2) is missing the Chain ID and Network fields, which would give the user more context to make their decision.

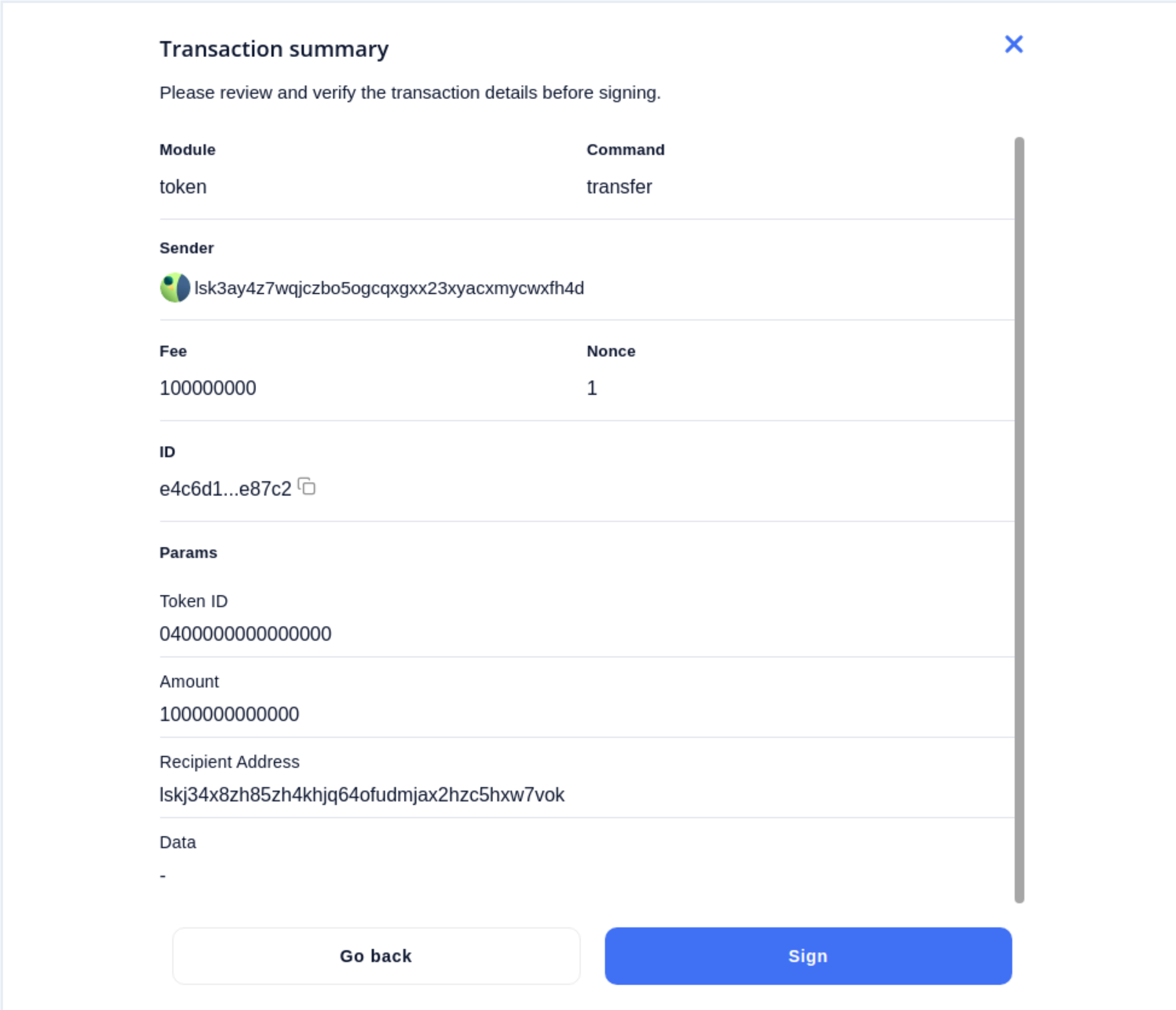

Figure 83.2: Second screen of the Signature Request process

For the second example in `lisk-mobile`, the chain ID used to encode and sign transactions (the `_networkStatus.chainID` variable in figure 83.3) comes from the online service (figures 83.4 and 83.5). The `chainID` is neither validated in code for being the expected length, nor shown to the user for confirmation.

```
signedTx = Lisk.transactions.signTransaction(
  this.transaction,
  Buffer.from(this._networkStatus.chainID, 'hex'),
  Buffer.from(privateKey, 'hex'),
  this._paramsSchema
);
```

Figure 83.3: The transaction is signed with the `chainID` coming from the `_networkStatus` variable.
(`lisk-mobile/src/modules/Transactions/utils/Transaction.js#187–192`)

```
const {
  data: networkStatusData,
  isSuccess: isNetworkStatusSuccess,
  isError: isErrorOnNetworkStatus,
} = useNetworkStatusQuery();
```

Figure 83.4: The `_networkStatus` variable is the same as the `networkStatusData` variable. (`lisk-mobile/src/modules/Transactions/hooks/useCreateTransaction.js#33–37`)

```
const config = {
  url: `${API_URL}/network/status`,
  method: 'get',
  event: 'get.network.status',
  ...customConfig,
};
```

Figure 83.5: The `networkStatusData` variable comes from a query to `lisk-service`. (`lisk-mobile/src/modules/Network/api/useNetworkStatusQuery.js#17–22`)

Exploit Scenario 1

A user clicks "Continue" on the first screen of the signature request modal without too much thought because he wants to see the other transaction details, including the amount of tokens to be transferred and the destination address. The user misses that the chain ID was not the one he expected. On the second screen, the user validates that all the data is as expected. The user approves the transaction with the incorrect chain ID.

Exploit Scenario 2

Alice uses the `lisk-mobile` application to create a transaction. A malicious `lisk-service` provides her with a malicious chain ID. The transaction created and signed by Alice is destined to a different chain than she wanted.

Recommendations

Short term, modify the form to show the Chain ID and Network fields in the transaction approval screen in `lisk-desktop`. Have the code validate the chain ID and have the form show it in `lisk-mobile`. This will ensure that the user has all the information they need to make an informed decision. Research security implications of using various data received from external sources without validation. For example, not validating or checking the integrity of received schemas or token metadata (e.g., denomination and decimals) may have a critical impact on the security of mobile and desktop wallets.

Long term, list all information coming from online services and ensure that it is validated in the code. Then decide if it should be shown to a user for manual validation and confirmation. If not, document that this particular data is safe to be used without the user's knowledge and interaction.

Moreover, test ways that data could be hidden via malicious input—for example, through a very large transaction parameter (see also TOB-LISK-90).

84. Users can be tricked into unknowingly authorizing a dapp on a chain ID	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK-84
Target: `lisk-desktop` WalletConnect integration	

Description

The `ConnectionSummary` component—which is responsible for showing the user all the details from a decentralized application's (dapp's) WalletConnect connection request—shows the user only the first chain ID from the request, even if several chain IDs are part of the request.

```
const application = {
  data: {
    name: proposer.metadata.name,
    projectPage: proposer.metadata.url.replace(/\/$/, ''),
    icon: proposer.metadata.icons[0],
    address: `Chain ID: ${requiredNamespaces.lisk.chains[0].replace('lisk:', '')}`,
  },
};
```

Figure 84.1: lisk-desktop/src/modules/blockchainApplication/connection/components/ConnectionSummary/index.js#49–56

As figure 84.1 shows, the code sets the `address` field to only the first chain ID instead of to all the chain IDs in the request. The `address` field is what the UI will show to the user, so all other chain IDs in the request will be hidden.

Later, the code will call the `client.approve` function with all the chain IDs that it received, as highlighted in figure 84.2.

```
export const onApprove = async (proposal, selectedAccounts) => {
  const { id, params } = proposal;
  const { requiredNamespaces, relays } = params;

  // Normalize the information according to requirements of the bridge
  const namespaces = Object.entries(requiredNamespaces).reduce((namespace, [key,
value]) => {
    const accounts = value.chains
      .map((chain) => selectedAccounts.map((account) => `${chain}:${account}`))
      .flat();
```

```
    namespace[key] = {
      accounts,
      ...value,
    };

    return namespace;
  }, {});

  const [err, response] = await to(
    client.approve({
      id,
      relayProtocol: relays[0].protocol,
      namespaces,
    })
  );
```

Figure 84.2: `lisk-desktop/libs/wcm/utils/sessionHandlers.js#13–37`

Exploit Scenario

A malicious dapp makes a WalletConnect connection request for two chain IDs: a sidechain and the mainchain. Lisk's desktop wallet UI shows only the first chain ID, the sidechain ID, which the user is happy to accept. Without the user's knowledge, the mainchain ID is also approved. Later, the malicious dapp makes a request to transfer tokens on the mainchain. The user approves the transaction because he thinks that the dapp can request transactions only on the sidechain, and he fails to observe that the chain ID was changed.

Recommendations

Short term, change the code to display all chain IDs being requested in the WalletConnect authorization proposal. Alternatively, have the code reject requests that contain more than one chain ID.

88. Missing round-trip property between parseSearchParams and stringifySearchParams

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-88
Target: `lisk-desktop`	

Description

The code uses the `parseSearchParams` and `stringifySearchParams` functions to parse URL search parameters. These functions should be round trip—i.e., `parseSearchParams(arg)` should always equal `parseSearchParams(stringifySearchParams(parseSearchParams(arg));` however, this is not the case. The fuzzer shown in figure 88.1 identifies several situations where the round-trip property fails. For example, an ampersand [&] character in a parameter value will be stringified without encoding, which will cause a later parsing to use it as a parameter separator.

```
function (data /*: Buffer */) {
    const fuzzerData = data.toString();

    const originalParams = `?x=${encodeURIComponent(fuzzerData)}`
    // const originalParams = `?x=${encodeURIComponent("2&b=2&c=3,4,5")}`

    const originalParamsParsed = parseSearchParams(originalParams);
    const originalParamsStringified = stringifySearchParams(originalParamsParsed);
    const newParamsParsed = parseSearchParams(originalParamsStringified);

    if (Object.keys(originalParamsParsed).length !=
Object.keys(newParamsParsed).length) {
        throw new Error(`originalParamsParsed !== newParamsParsed:
'${JSON.stringify(originalParamsParsed)}' !==
'${JSON.stringify(newParamsParsed)}'`);
    }

    // TODO: Do a deep comparison of the originalParamsParsed and newParamsParsed
objects
};
```

Figure 88.1: Fuzzer to detect round-trip issues between the `parseSearchParams` and `stringifySearchParams` functions

The severity of this finding is set to undetermined because while attacker-controlled data may reach these functions (e.g., through the `lisk://` protocol), we did not find a direct exploitation path.

Recommendations

Short term, fix the `parseSearchParams` and `stringifySearchParams` functions and run the fuzzer above to stress test the fixes.

Long term, consider whether manual parsing of URLs is really necessary. Almost always, the correct option is to use JavaScript's URL object instead.

89. Several lisk-desktop identifiers are not unique	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-89
Target: `lisk-desktop`	

Description

The `lisk-desktop` application stores names for accounts, networks, and bookmarks, but these are not all guaranteed to be unique. As a result, users may end up using the wrong account for a transaction. Figure 89.1 shows this issue for bookmarks.

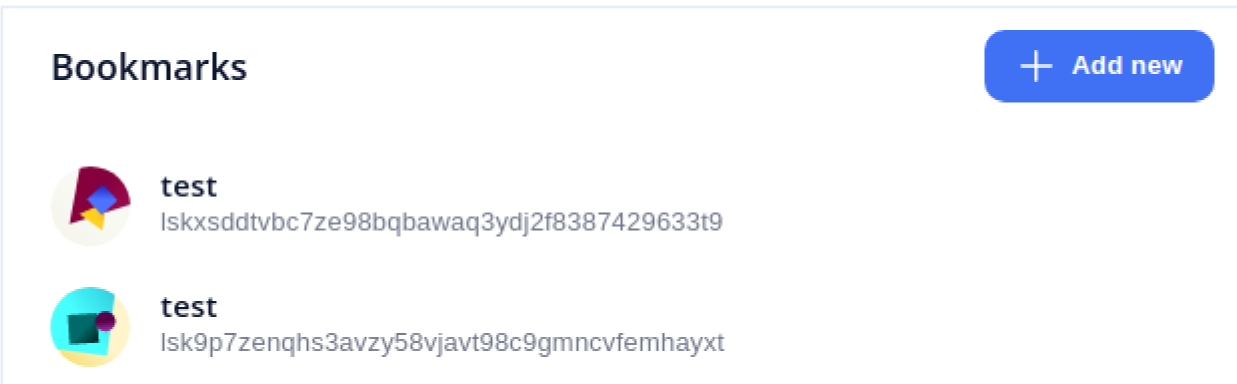

Figure 89.1: Two bookmarks with the same identifier

Exploit Scenario 1

A user bookmarks two accounts with the same name by mistake. When choosing which one to send tokens to, they choose the wrong one.

Exploit Scenario 2

A user stores two wallets with the same name by mistake. When choosing which one to send tokens from, they choose the wrong one.

Recommendations

Short term, add a check to guarantee that each of the account, network, and bookmark identifiers is unique. This will reduce the likelihood of user mistakes.

Long term, review any other identifiers stored in `lisk-desktop`, and if it makes sense, ensure they are unique.

93. Incorrect entropy in lisk-desktop passphrase generation

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-93
Target: `lisk-desktop` passphrase generation	

Description

The `lisk-desktop` application generates a mnemonic (also called a passphrase) by first generating a seed and then calling the `mnemonic` function of the `bitcore-mnemonic` library, as shown in figure 93.1. The seed generation code is incorrect because it uses the incorrect amount of entropy. It is also unnecessarily complex and hard to read.

```
/**
 * Generates a passphrase from a given seed array using mnemonic
 *
 * @param {string[]} seed - An array of 16 hex numbers in string format
 * @returns {string} The generated passphrase
 */
export const generatePassphraseFromSeed = ({ seed }) =>
  // eslint-disable-next-line no-buffer-constructor
  new mnemonic(Buffer.from(seed.join(''))).toString();

/**
 * Generates a random passphrase using browser crypto api
 *
 * @returns {string} The generated passphrase
 */
export const generatePassphrase = () => {
  // istanbul ignore next
  const crypotObj = window.crypto || window.msCrypto;
  return generatePassphraseFromSeed({
    seed: [...crypotObj.getRandomValues(new Uint16Array(16))].map((x) =>
      `00${(x % 256).toString(16)}`.slice(-2)
    ),
  });
};
```

Figure 93.1: Code that generates a passphrase in `lisk-desktop`
(`lisk-desktop/src/modules/wallet/utils/passphrase.js#78–101`)

The seed is 16 bytes long, which is 16 * 8 = 128 bits of entropy. As specified in BIP-0039 and shown in figure 93.2, this is enough entropy to generate a 12-word mnemonic; however, the code generates a 24-word mnemonic, which should require 256 bits of entropy.

```
|  ENT   | CS | ENT+CS |  MS  |
+--------+----+--------+------+
|  128   |  4 |   132  |  12  |
|  160   |  5 |   165  |  15  |
|  192   |  6 |   198  |  18  |
|  224   |  7 |   231  |  21  |
|  256   |  8 |   264  |  24  |
```

Table 93.1: The initial entropy length (ENT), the checksum length (CS), and the length of the generated mnemonic sentence (MS) in words (https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki#generating-the-mnemonic)

The error occurs when calling the `Buffer.from` function, which transforms each character of the hex-encoded seed into a byte instead of transforming each sequence of two characters into a byte. For example, in figure 93.2, we show how the character `d` is transformed into the `0x64` byte rather than the `d1` characters being transformed into the `0xd1` byte. This results in a `Buffer` with 256 bits but only 128 bits of entropy.

```
const seed = [...crypto.getRandomValues(new Uint16Array(16))].map((x) => `00${(x %
256).toString(16)}`.slice(-2)).join('')
console.log(seed)
// E.g., d19372f91b24bcb0d72f97b1049631ab
console.log(Buffer.from(seed))
// E.g., <Buffer 64 31 39 33 37 32 66 39 31 62 32 34 62 63 62 30 64 37 32 66 39 37
62 31 30 34 39 36 33 31 61 62>
```

Figure 93.2: Example code that shows the entropy generation problems

In contrast, `lisk-mobile` uses `lisk-passphrase`, which in turn uses the `bip39` library for mnemonic generation.

Exploit Scenario

A user uses `lisk-desktop` to generate a passphrase with a perceived strength of 256 bits. The code uses only 128 bits of entropy to generate the 24 words. The user's key is generated with at most 128 bits of strength—or potentially less if any other attacks apply against this specific flaw. An attacker discovers an attack against this flaw, recovers the user's keys, and steals their funds.

Recommendations

Short term, modify `lisk-mobile`'s code to use `lisk-passphrase`'s implementation instead of the current implementation. This will keep the passphrase generation code centralized, which makes it easier to audit and reduces the likelihood of bugs such as this one being introduced.

Long term, review other cryptographic-related code and ensure that there are not multiple implementations of the same functionality or any duplicated code.

94. lisk-desktop attempts to open a nonexistent modal

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-94
Target: `lisk-desktop`	

Description

The `ConnectionSummary` component tries to open the `connectionSuccess` modal, which does not exist. The developer likely intended for this component to open the `connectionStatus` modal.

```
addSearchParamsToUrl(history, {
  modal: 'connectionSuccess',
  action: ACTIONS.REJECT,
  status: result.status,
  name: result.data?.params?.proposer.metadata.name ?? '',
});
```

Figure 94.1: *`lisk-desktop/src/modules/blockchainApplication/connection/components/ConnectionSummary/index.js#35–40`*

Recommendations

Short term, fix the issue by having the `ConnectionSummary` component open the correct modal.

Long term, instead of hard coding a modal's name when opening it, use constants based on the modals defined in the `routes.js` file. This will remove the possibility of bugs such as the one in this finding and improve the code quality.

95. Phishing risk on the deviceDisconnectDialog modal	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-95
Target: `lisk-desktop`	

Description

Part of the error message shown in the `deviceDisconnectDialog` modal is controlled by the `model` URL parameter. Developers expected this field to receive only the name of a device that was disconnected, but it can receive any message, including a phishing message, as shown in figure 95.1.

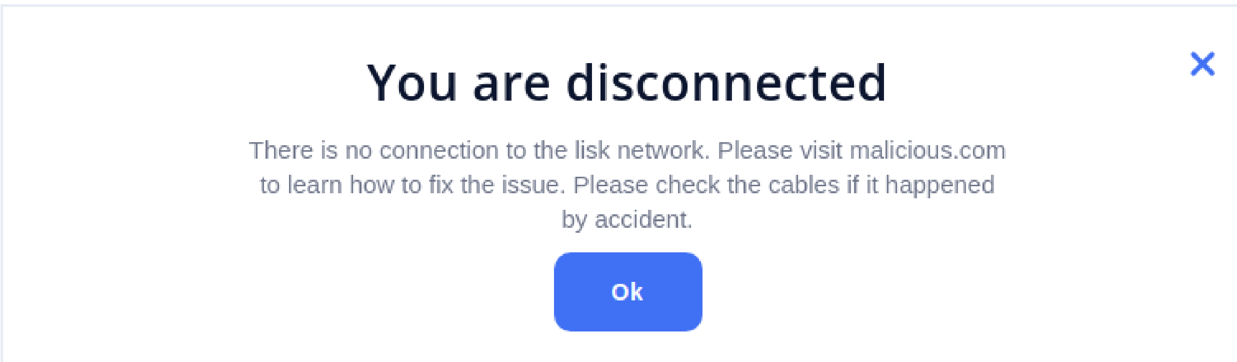

Figure 95.1: Error message with a phishing message from an attacker

This message can be triggered by navigating to the following URL:

```
/?modal=deviceDisconnectDialog&model=lisk network. Please visit
malicious.com to learn how to fix the issue
```

Exploit Scenario

An attacker sends a `lisk://` protocol link to the URL above to a target user. The user clicks the link and is shown the error message. He follows the instructions in the phishing message, inserts his passphrase in the attacker's website, and loses his funds.

Recommendations

Short term, hard code the list of supported devices that can be shown in the URL. Alternatively, if the list of devices is unknown, limit the size of the message.

96. Use of JavaScript instead of TypeScript	
Severity: **Low**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-LISK-96
Target: `lisk-desktop`	

Description

The `lisk-desktop` application is developed in JavaScript instead of TypeScript. TypeScript is a strongly typed language that compiles to JavaScript and allows developers to specify the types of variables and function arguments; it fails to compile if there are type mismatches. Contrarily, JavaScript code will crash (or worse) during runtime if there are type mismatches.

In summary, TypeScript is preferred over JavaScript for the following reasons:

- It improves code readability; developers can easily identify variable types and the types that functions receive.
- It improves security by providing static type checking that catches errors during compilation.
- It improves support for integrated development environments (IDEs) and other tools by allowing them to analyze the types of variables.

Figure 96.1 shows an example of code in `lisk-desktop` that compiles even though it is dangerously wrong. The `signTransactionWithPrivateKey` function is called with the arguments in the wrong order (figure 96.2 shows the correct order), which should cause the call to fail, or worse, perform the incorrect cryptographic operation.

```
const res = transactions.signTransactionWithPrivateKey(
  schema,
  transaction,
  Buffer.from(networkIdentifier),
  this.keypair.secretKey
);
```

Figure 96.1: Incorrect argument types for the `signTransactionWithPrivateKey` function (`lisk-desktop/libs/wcm/utils/requestHandlers.js#28–33`)

```
export declare const signTransactionWithPrivateKey: (
    transactionObject: Record<string, unknown>,
    chainID: Buffer,
    privateKey: Buffer,
    paramsSchema?: object) => Record<string, unknown>;
```

Figure 96.2: The correct argument types for the `signTransactionWithPrivateKey` function

This issue with `signTransactionWithPrivateKey` is not set to a higher severity because it is dead code that is never called.

Exploit Scenario

A type mismatch bug in `lisk-desktop` is missed, and the code is deployed to production. The bug causes the application to incorrectly perform a cryptographic operation (as in the example above). A user's passphrase is incorrectly generated and an attacker can generate it on their own to steal the user's funds.

Recommendations

Short term, rewrite newer parts of the application in TypeScript. TypeScript can be used side by side with JavaScript in the same application, which allows it to be introduced gradually.

Long term, gradually rewrite the entire application in TypeScript when modifying old JavaScript code.

Detailed Findings—Lisk Mobile

68. Mobile iOS application does not use system-managed login input fields	
Severity: **Low**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-LISK-68
Target: `lisk-mobile`, iOS	

Description

The `lisk-mobile` application does not set the type for username and password fields. Since iOS 12, the iOS software development kit (SDK) has included text field properties to automate the process of password generation and credential entry, offering to auto-generate strong passwords and save them in the system keychain or a password manager.

Furthermore, identifying these fields as login input fields may help prevent entered text from being misused by iOS. Text entered into fields that lack these identifiers may be sent to a spell-check service, added to an autocomplete dictionary, or otherwise cached in a way that increases their risk of exposure.

Exploit Scenario

Bob installs `lisk-mobile` on his iOS phone and cannot use a machine-generated password because `lisk-mobile` does not use system-managed login input fields on iOS. Bob chooses an insecure password, which is cached in an autocomplete dictionary by iOS.

Recommendations

Short term, consider using the `UITextContentType` property introduced in iOS 12 to identify username and password fields, allowing automated, strong password generation. Make sure the generated password will not be saved in the keychain while bypassing the `lisk-mobile` custom login for handling passwords.

Long term, stay abreast of new security features added to the iOS SDK.

References

- textContentType, Apple Developer
- About the Password AutoFill workflow, Apple Developer
- `textContentType`, React Native

69. Mobile iOS application does not exclude keychain items from online backups	
Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-LISK-69
Target: `lisk-mobile`, iOS	

Description

The `lisk-mobile` application does not prohibit its keychain items from being saved to an iTunes backup or uploaded to iCloud. Both Apple, Inc. and any attacker with access to a user's iTunes or iCloud backups will have access to that user's private data.

Exploit Scenario

Alice gains physical access to Bob's phone and knows his passcode. She initiates a backup of Bob's phone to iTunes and extracts all the `lisk-mobile` sensitive keychain data.

Alternatively, Alice identifies user email addresses and then uses a previously disclosed password database to guess the users' current iCloud passwords. She retrieves iCloud backups that contain sensitive `lisk-mobile` keychain data for a large number of users.

Recommendations

Short term, explicitly set a `ThisDeviceOnly` accessibility class (e.g., `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` or `WHEN_UNLOCKED_THIS_DEVICE_ONLY`) for all keychain items. This should prevent keychain data from being migrated to iTunes and iCloud backups.

Long term, empirically validate that no sensitive data is stored to a backup of `lisk-mobile`.

References

- Keychain Services, Apple Developer
- `Keychain.ACCESSIBLE` enum, Keychain/Keystore Access for React Native

70. Mobile iOS application does not disable custom iOS keyboards	
Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-LISK-70
Target: `lisk-mobile`, iOS	

Description

The `lisk-mobile` application does not disable custom keyboards. Since iOS 8, users have been able to replace the system's default keyboard with custom keyboards that can be used in any application. Custom keyboards can—and very frequently do—log and exfiltrate the data that users enter.

Custom keyboards are not enabled when users type into secure fields (e.g., a password field). However, `lisk-mobile` has a feature that allows copying secret recovery phrases to a clipboard, which does not benefit from the same protection as secure fields.

Moreover, custom keyboards could log all of a user's keystrokes in regular fields, such as those where users type their personal information.

Exploit Scenario

Alice creates a custom keyboard that Bob uses. Alice's keyboard silently exfiltrates all of Bob's keystrokes in the `lisk-mobile` application. Because Bob copied his secret recovery phrases at one point when using the application, Alice steals his private keys and all his funds.

Recommendations

Short term, disable third-party keyboards within `lisk-mobile` to prevent disclosure of sensitive data entered by the user. Third-party keyboards can be disabled globally by adding the `application:shouldAllowExtensionPointIdentifier:` method to the application client's `UIApplicationDelegate` object.

Long term, review other clipboard-related issues in iOS.

References

- Przemyslaw Samsel, "Third-party iPhone keyboards vs your iOS application security," Securing, October 26, 2022.

71. Mobile application uses invalid KDF algorithm and parameters	
Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-71
Target: `lisk-mobile`	

Description

The `lisk-mobile` application uses the PBKDF2 algorithm, instead of the recommended argon2id algorithm, to derive the encryption key from a password. Moreover, the `memorySize` parameter would be incorrect if the argon2id algorithm was used. The issue is similar to TOB-LISK-12.

```
const crypto = await encrypt.encryptMessageWithPassword(plainText, password, {
  kdf: 'PBKDF2',
  kdfparams: {
    parallelism: 4,
    iterations: 1,
    memorySize: 2024,
  },
});
```

Figure 71.1: Vulnerable part of the `encryptAccount` method (`lisk-mobile/src/modules/Auth/utils/encryptAccount.js#27–34`)

Exploit Scenario

An attacker gains access to a file with an encrypted recovery phrase. The attacker cracks the encryption easily because the PBKDF2 algorithm with only a single iteration is a weak protection.

Recommendations

Short term, replace the PBKDF2 algorithm with argon2id, which is more secure, and set the `memorySize` argument to `2097152` or another value chosen from RFC 9106. If the PBKDF2 algorithm is the intended one, remove the unnecessary `memorySize` and `parallelism` parameters (which are not used by the algorithm) and set the `iterations` parameter to a recommended value.

References

- How PBKDF2 strengthens your 1Password account password, 1Password

72. Mobile iOS application includes redundant permissions

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-72
Target: `lisk-mobile`, iOS	

Description

The `Info.plist` file includes unnecessary descriptions for various permissions, as shown in figure 72.1. Most of the referenced permissions are not used by the `lisk-mobile` application. Including descriptions for unused permissions may unnecessarily alarm users, who may incorrectly think that `lisk-mobile` gathers data (e.g., location data).

```
<key>NSLocationAlwaysUsageDescription</key>
<string>This is a standard App Store requirement. Lisk Does not access this
feature.</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>This is a standard App Store requirement. Lisk Does not access this
feature.</string>
```

Figure 72.1: Example permission descriptions in `Info.plist` (`lisk-mobile/ios/Lisk/Info.plist#54–57`)

The unnecessary descriptions are supposed to be required by the App Store reviewers. However, only descriptions for permissions that are linked to an application binary are required. We recommend not linking the `lisk-mobile` binary with unused permissions. For example, the `CLLocationManager` symbol, which is required for the `NSLocationAlwaysUsageDescription` key, is included in the binary via the Core Location dynamic library (`dylib`), which is probably included by the `react-native-permissions` package.

```
$ xcrun dyldinfo -bind ./Lisk | grep Location
__DATA  __got            0x100440E60    pointer      0 UIKit
_UITextContentTypeLocation
__DATA  __objc_classrefs 0x10050EAD0    pointer      0 CoreLocation
_OBJC_CLASS_$_CLLocationManager
```

Figure 72.2: Discovering symbols included (linked) in a binary

React Native packages should be configurable so that the final application is built without links to unused permissions.

Moreover, there are indications of missing required permission descriptions. For example, use of the `_setsockopt` and `performMulticastRequest` methods in the Lisk binary indicates that the `NSLocalNetworkUsageDescription` key may be required.

```
$ nm ./Lisk | rg _setsockopt
                 U _setsockopt
$ strings ./Lisk | rg performMulticastRequest
performMulticastRequest:forGroup:onInterface:error:
```

Figure 72.3: Discovering methods used in a binary

Recommendations

Short term, discover which React Native packages are causing the inclusion of symbols that imply requirements for unused permissions. Configure the packages to avoid including unnecessary symbols in the Lisk application binary. If the desired configuration is not possible, contact the maintainers of the problematic packages and work with them to enable the desired configurations.

Consider using `UIImagePickerController` and `PHPicker` APIs for reading QR codes. These APIs may allow removing the `NSPhotoLibraryUsageDescription` and `NSPhotoLibraryAddUsageDescription` permissions from the `Info.plist` file, which would increase users' trust in the `lisk-mobile` application.

Long term, review the Android permissions used by `lisk-mobile`. Decide whether specific permissions can be removed, and if so, remove them. For example, the `CHANGE_WIFI_MULTICAST_STATE` and `READ_EXTERNAL_STORAGE` permissions seem to be overly broad. Instead of broad filesystem permissions, consider using Scoped Storage.

74. Mobile iOS application disables ATS on iOS devices

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-74
Target: `lisk-mobile`, iOS	

Description

The `lisk-mobile` application explicitly disables App Transport Security (ATS), a network security feature on Apple platforms that improves the use of encryption and integrity protections for network communications. It does this by requiring network connections to be secured by Transport Layer Security (TLS) with stronger-than-default certificates and ciphers. ATS blocks connections that fail to meet the minimum security requirements.

```
<key>NSAppTransportSecurity</key>
<dict>
	<key>NSAllowsArbitraryLoads</key>
	<true/>
	<key>NSExceptionDomains</key>
	<dict>
		<key>localhost</key>
		<dict>
			<key>NSExceptionAllowsInsecureHTTPLoads</key>
			<true/>
		</dict>
	</dict>
</dict>
```

Figure 74.1: Current `NSAppTransportSecurity` settings (`lisk-mobile/ios/Lisk/Info.plist#27–39`)

By default, all TLS connections on iOS check that the server certificate adheres to the following requirements:

- It has an intact digital signature.
- It is not expired.
- It has a name that matches the server's DNS name.
- It is signed by a certificate chain ending in a valid certificate authority.

ATS requires these checks and provides the following additional checks:

- The server certificate must be signed with an RSA key of at least 2,048 bits or an ECC key of at least 256 bits.
- The server certificate must use SHA-2 with a digest length of at least 256 bits.
- The connection must use TLS v1.2 or later.
- Data must be exchanged using AES-128 or AES-256.

The link must support perfect forward secrecy (PFS) through an elliptic curve Diffie–Hellman ephemeral (ECDHE) key exchange.

Exploit Scenario

Alice logs in to `lisk-mobile` via network communications encrypted with an outdated version of TLS and weak ciphers. Bob is a network administrator at an intermediate routing point with access to Alice's network traffic. He uses an active attack against the outdated version of TLS to decrypt Alice's traffic, or he collects it for future decryption.

Recommendations

Short term, precisely define the ATS exceptions required for `lisk-mobile`. Configure ATS exceptions only when needed, use the narrowest possible exception, and upgrade `lisk-mobile` servers to meet the requirements imposed by ATS.

Long term, remove all exceptions. All network communications should meet the minimum requirements imposed by ATS.

References

- `NSAllowsArbitraryLoads`, Apple Developer
- Preventing Insecure Network Connections, Apple Developer
- RFC 7457: Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS), February 2015.

75. Mobile application does not implement certificate pinning	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LISK-75
Target: `lisk-mobile`	

Description

The `lisk-mobile` application does not enforce validation of HTTPS connections through certificate pinning.

Certificate pinning is a method of allowlisting a specific server certificate within an application to reduce the likelihood of an attacker intercepting the communications. When making a connection to the back-end API, if the certificate presented by the server does not match the signature of the pinned certificate, the application can infer that the path has been modified and terminate operations.

Due to the high complexity of preparing this attack, an attacker could target only a small number of users. After a successful attack, a single device's HTTPS communications to and from the back-end API would be exposed to the attacker.

Exploit Scenario

An attacker with local access to the target device could add a certificate to the trust store, allowing a proxy to decrypt the HTTPS communications. All data sent over that connection would be exposed without the user seeing any application warnings.

Recommendations

Short term, configure the application to function only with a known-good certificate presented from the intended back-end API. Start by implementing the pinning mechanism for back ends that are controlled by the Lisk team, especially `lisk-service` instances. If `lisk-mobile` detects that a wrong certificate or public key is used, it should alert the user and reject the connection. The issue would indicate a serious problem in the system, and users should be instructed to take actions like consulting community social media or updating the application.

Later, implement the pinning mechanism for dynamic services (i.e., sidechain applications that a user can register in the application). Since these applications are not controlled by the Lisk team, a mechanism for establishing valid certificates or public keys must be designed. For example, `lisk-mobile` may trust and save locally a public key received during the first connection (during registration) and then use that key to validate all further connections. If a connection is detected to use another certificate or key than the trusted one, the user should be alerted and either the connection should be rejected, or the user should be informed about the risks and asked to accept the new certificate or key. For this mechanism to be effective, owners of the sidechain applications must maintain a stable TLS certificate or key to avoid spamming users with false alarms.

Long term, implement unit tests to validate that only the pinned certificate is being accepted by the application.

References

- Certificate and Public Key Pinning Control, OWASP
- TrustKit
- TrustKit Android

76. Mobile application biometric authentication is prone to bypasses	
Severity: **High**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-LISK-76
Target: `lisk-mobile`, iOS	

Description

The `lisk-mobile` application allows users to store passwords in the iOS Keychain and authorize access to the stored passwords with biometric authentication. However, the authorization is implemented at the application level instead of the Keychain level. Therefore, it is prone to bypasses: users may access plaintext passwords without biometric authentication in some cases.

The biometric authentication is performed by the `bioMetricAuthentication` function, which is called after a password is retrieved from the Keychain (figure 76.1).

```
const tryFetchAccontPasswordFromBiometrics = async () => {
  if (sensorType) {
    const accountPassword = await fetchAccountPassword();
    if (accountPassword) {
      bioMetricAuthentication({
        successCallback: () => {
          onSubmit(accountPassword);
        },
      });
    }
  }
};
```

Figure 76.1: The `fetchAccountPassword` method retrieves the password from the Keychain before the user is asked for biometric authentication. (*`lisk-mobile/src/modules/Auth/components/PasswordForm/index.js#32–43`*)

The Keychain itself does not require biometric authentication because the `setGenericPassword` method is called without any access control option (figure 76.2).

```
await setGenericPassword(uniqueId, JSON.stringify(deviceAccounts), {
  accessGroup: '58UK9RE9TP.io.lisk.mobile',
  service: 'io.lisk.mobile',
});
```

Figure 76.2: Passwords (`deviceAccounts`) are stored in the Keychain without a biometric authentication requirement. (`lisk-mobile/src/modules/Auth/utils/recoveryPhrase.js#121–124`)

Exploit Scenario

Bob steals Alice's mobile device while it is in an unlocked state. He accesses the Keychain directly and retrieves all `lisk-mobile` passwords stored in it. He uses the passwords to decrypt encrypted recovery phrase files and then steals all Alice's funds.

Recommendations

Short term, reimplement biometric authentication so that it is enforced on the Keychain level. Accessing passwords stored in the Keychain should not be possible without biometric authentication, even outside the `lisk-mobile` application. Have the code use `react-native-keychain`'s `BIOMETRY_CURRENT_SET` flag to allow access to passwords only with already enrolled fingerprints. This will prevent attackers from enrolling their own fingerprints and using them to access passwords.

Long term, review `lisk-mobile` authentication mechanisms to minimize the impact of an unauthorized user who gains physical access to a device. Ensure that authentication is required for every sensitive functionality. For example, disallowing a transition to the next page in the user interface could be bypassed, but encrypting a shared secret key with a required password would require any attacker to authenticate accordingly.

References

- Note regarding temporariness of keys in the Keychain, "Local Authentication on iOS," OWASP

77. Mobile application is susceptible to URI scheme hijacking due to not using Universal Links and App Links	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK-77
Target: `lisk-mobile`	

Description

The `lisk-mobile` application defines the `lisk://` URI scheme for receiving messages from other apps on the device. URI schemes can be hijacked by another app if the malicious app registers the same scheme and is also installed on the device. Consequently, a rogue app could receive messages sent via URI schemes intended for `lisk-mobile`.

More secure linking features are Universal Links and App Links (for iOS and Android applications, respectively). These links are bound to a web domain, making it impossible for a malicious application to register a domain that belongs to another application.

Exploit Scenario

Mallory creates a malicious application that mimics `lisk-mobile` and registers the same `lisk://` URI scheme. Bob installs it. Alice, a trusted actor, then sends Bob a link for a transaction with the `lisk://` scheme. Bob clicks on it and is redirected to the malicious application. Bob is unable to distinguish the malicious application from the true one and imports his blockchain account. Mallory steals his key and all his funds.

If Universal Links and App Links were used by `lisk-mobile`, Bob would not be able to open the link provided by Alice in the malicious application and would have a chance to detect the attack.

Recommendations

Short term, remove support for custom URL schemes and support only Universal Links and App Links. Implement procedures for proving ownership of the domain used for linking and for keeping the domain available (i.e., preventing domain hijacking attacks).

Long term, add support for safety features of iOS and Android such as Play Integrity API and DCAppAttestService.

References

- Support Universal Links, Apple Developer
- Handling Android App Links, Android Developer

78. Mobile iOS application filesystem encryption is not enabled for locked devices	
Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-LISK-78
Target: `lisk-mobile`, iOS	

Description

iOS has a data protection feature providing automatic encryption of files stored in the filesystem. By default, files are encrypted only until the iOS device is unlocked the first time after booting. The most secure setting of data protection is to always encrypt files when the device is locked and decrypt them only after it is unlocked. Lisk should configure data protection with the most secure settings for files used by Lisk.

Exploit Scenario

Alice steals Bob's iOS phone and exploits a vulnerability in the iOS screen lock functionality. She gains access to the filesystem and steals all the files, which are not encrypted by iOS. Among the stolen files are those containing recovery phrases encrypted with Bob's password. Alice exploits the vulnerability described in TOB-LISK-76 to learn the password, decrypts the recovery phrases, and steals Bob's funds.

Recommendations

Short term, add the `com.apple.developer.default-data-protection` key with the `NSFileProtectionComplete` value to `lisk-mobile` entitlements.

Long term, review all security features and configuration options provided by the iOS system, and implement relevant ones in Lisk.

79. Mobile Android application permission riding is possible	
Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-LISK-79
Target: `lisk-mobile`, Android	

Description

Users of the `lisk-mobile` application grant permissions to the application, trusting the Lisk team to use the permissions only for fair purposes. However, third-party dependencies of `lisk-mobile` may silently misuse privileges granted to `lisk-mobile` and perform malicious operations.

To prevent third-party dependencies from misusing the permissions a user conceded to an application (a permission riding attack), Android introduced the data access auditing feature. This feature enables developers to monitor and limit the use of permissions by an application's dependencies.

Exploit Scenario

The maintainers of a dependency used by `lisk-mobile` are malicious. They update their dependency code to steal users' data. The `lisk-mobile` application is updated to a new version, which includes the malicious third-party code. The malicious code is allowed to use all permissions granted to `lisk-mobile`.

Recommendations

Short term, research and use the data access auditing feature to prevent third-party dependencies from misusing permissions granted to `lisk-mobile`.

Long term, research whether similar features are available for iOS applications.

87. Mobile application caches password in transaction-signing form	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Exposure	Finding ID: TOB-LISK-87
Target: `lisk-mobile`	

Description

The `lisk-mobile` application stores a user's password in process memory during the transaction-signing process. The behavior can be observed by taking the following steps:

- Create a new, invalid transaction in the “Send token” form. To make the transaction invalid, set the “Amount” to a larger value than the available balance.
- Click the “Continue” button and then the “Send” button.
- Provide the password.
- The transaction will fail, as shown in the left screenshot of figure 87.1.
- Click the “Try again” button.
- Change the parameters of the transaction arbitrarily to make the transaction valid.
- Again click the “Continue” button and then the “Send” button.
- The password will be populated automatically and can be made visible by clicking the eye icon.

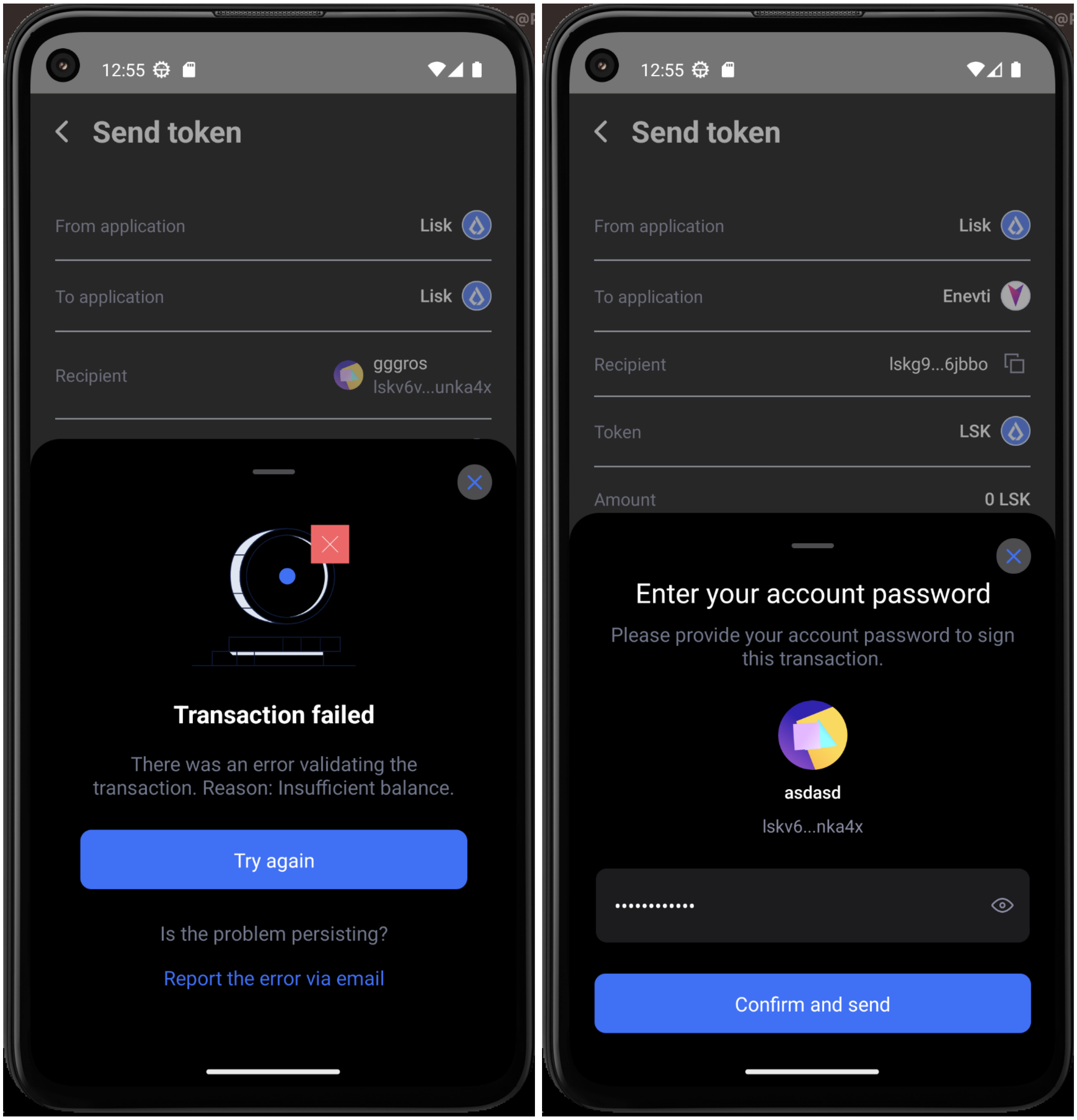

Figure 87.1: Two screenshots demonstrating that a password is cached until a transaction is successfully submitted or abandoned.

Exploit Scenario

Alice uses `lisk-mobile` to create a new transaction. She makes a typo and the "Transaction failed" dialog box appears. Suddenly, Bob grabs Alice's phone and runs. He can now read Alice's password and steal all her funds.

Recommendations

Short term, do not have the application cache passwords during transaction processing. Users should be required to provide the password again even after a retry of a failed transaction. To provide a better experience, have the code validate a transaction before asking the user for a password.

Long term, ensure that a password is never stored in application memory longer than necessary.

90. Mobile application insufficiently validates and incorrectly displays Amount value in transaction transfer

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK-90
Target: `lisk-mobile`	

Description

The `lisk-mobile` "Send token" feature consists of two screens. In the first one, a user provides values like an amount to send and a recipient address. The second screen displays all the provided data for confirmation. An attacker can perform a phishing attack by providing values that are different when shown on a phone screen and when used by the code.

One of the discovered attack vectors is related to the "Amount" field, which is expected to contain a number with an optional decimal separator. However, any data is accepted by the code.

A partial validation of the "Amount" field is done in the `useSendTokenAmountChecker` method (figure 90.1). There, the amount variable holds a user-provided string. The `isTransactionAmountValid` method checks the format of the string. If the amount is an invalid format, the code will set the `validatedAmount` variable to `0`, so the `isMaxAllowedAmountExceeded` variable is set to `true`.

```
const validatedAmount =
  selectedToken && isTransactionAmountValid(amount)
    ? BigInt(
        fromDisplayToBaseDenom({
          amount,
          displayDenom: selectedToken.displayDenom,
          denomUnits: selectedToken.denomUnits,
        })
      )
    : BigInt(0);

const isMaxAllowedAmountExceeded = maxAllowedAmount - validatedAmount <= 0;
```

Figure 90.1: An imprecise validation of the "Amount" field
(`lisk-mobile/src/modules/SendToken/hooks/useSendTokenAmountChecker.js#32–43`)

The issue—from a user perspective—is presented in figure 90.3. The user has opened the link shown in figure 90.2. In the first screenshot, the user sees an “Amount” of 1 LSK, but on the confirmation screen, the “Amount” shows the real number is 7331 LSK. It is unclear what number will be used by the code in the actual transaction-encoding process. In the worst case, the 7331 number will be used.

```
adb shell am start -a android.intent.action.VIEW -d
'lisk://wallet?amount=7331%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%
20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%
20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%201%20%20%20
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20%20%20%20'
```

Figure 90.2: adb command to open a malicious link in `lisk-mobile`

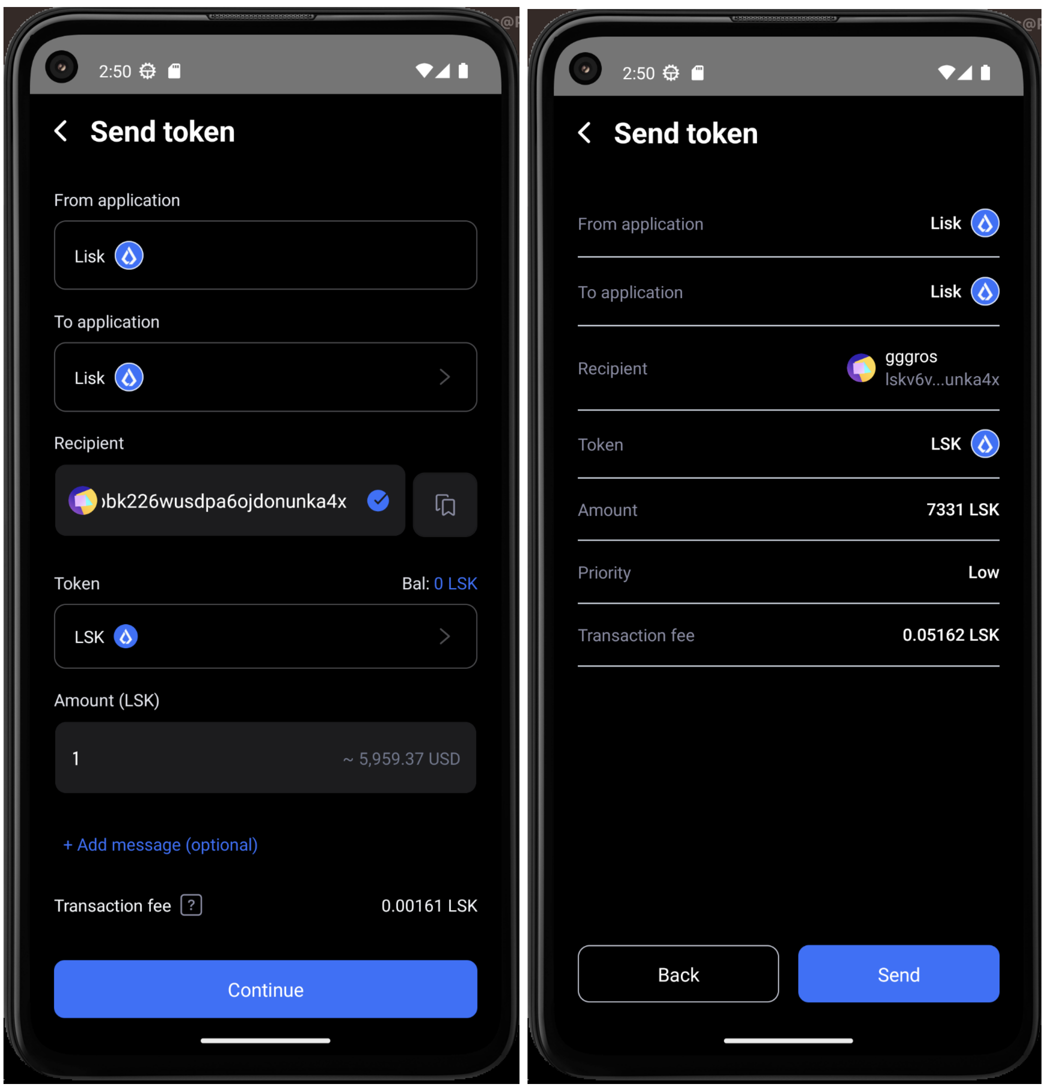

Figure 90.3: Two screenshots of `lisk-mobile` “Send token” feature

If a user changes the "Amount" field in any way or tries to manually type the same value, the application will reject the value as invalid.

The severity of the issue is set only to medium because, from limited testing, it appears that the invalid amount in the example would result in a zero value being used as the amount during transaction signing.

Exploit Scenario

An attacker sends a malicious `lisk://` link to a `lisk-mobile` user, asking the user to send him some small amount. The user opens the link and verifies the amount in the first screen. He ignores the USD estimation and does not verify the amount again in the confirmation screen. The user signs the transaction. The attacker receives a larger transfer than the user wanted to send.

Recommendations

Short term, improve validation of the "Amount" field to be stricter. Make sure that the user is always presented with the exact value that is used by the code.

Long term, perform root cause analysis of the issue and review similar attack vectors.

98. Mnemonic recovery passphrase can be copied to clipboard	
Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-LISK-98
Target: `lisk-mobile`, Android	

Description

The `lisk-mobile` application provides users with an option to copy newly generated secret recovery phrases (as mnemonic strings) to the device's clipboard. Storing secrets in a clipboard exposes them unnecessarily.

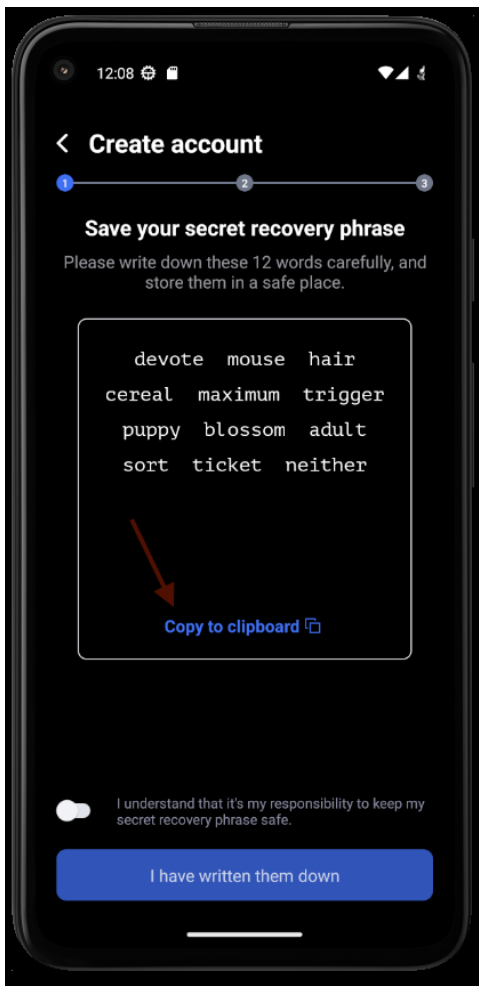

Figure 98.1: Screenshot with a new account recovery phrase generation and the "Copy to clipboard" button

The severity of the issue is set only to medium because new Android versions considerably limit the exploitability of bugs of this type.

Exploit Scenario

A user installs both `lisk-mobile` and a malicious application on his phone. He creates a new Lisk account, accidentally clicking on the "Copy to clipboard" button. The malicious application periodically scans the clipboard and thereby detects the mnemonic. The application sends the mnemonic to an online server. The user sends some funds to his new account, where the funds are stolen by the creator of the malicious application.

Recommendations

Short term, do not provide options for users to copy sensitive data like secret phrases. Use the `EXTRA_IS_SENSITIVE` flag for any sensitive data that must be copyable.

Long term, review the mobile application against other potential data exposure issues resulting from lack of clipboard use restrictions.

References

- "Be Aware of Copy and Paste," NowSecure
- Dmitrios Valsamara and Michael Peck, "Protecting Android clipboard content from unintended exposure," *Microsoft Security Blog*, March 6, 2023.

Detailed Findings—Lisk Service

63. ReDoS in API parameter validation	
Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-LISK-63
Target: Lisk Service API parameter validation	

Description

Several regular expressions used to validate API parameters in Lisk Service are vulnerable to regular expression denial-of-service (ReDoS) attacks. A regular expression vulnerable to ReDoS attacks is one that tests inputs exponentially more slowly in relation to the input length, which leads to resource exhaustion. The vulnerable patterns and inputs that can be used to exploit the regular expressions are listed below.

```
const MODULE = /^\b(?:[\w!@$&.]{1,32}|,)+\b$/;
```

Figure 63.1: `regex.MODULE` is vulnerable to ReDoS.
(`lisk-service/services/gateway/shared/regex.js#41`)

```
zx8dzx8dzx8dzx8dzxzx8dQ$1l88HY&M.{'
```

Figure 63.2: Input that causes ReDoS for `regex.MODULE`

```
const TOPIC = /^\b(?:(?:[0-9a-fA-F]{2,64}|lsk[a-hjkm-z2-9]{38}),?)+\b$/;
```

Figure 63.3: `regex.TOPIC` is vulnerable to ReDoS.
(`lisk-service/services/gateway/shared/regex.js#41`)

```
0202020202020202020202020202020202020202020202020202020202020x
```

Figure 63.4: Input that causes ReDoS for `regex.TOPIC`

```
blockID: { optional: true, type: 'string', min: 1, max: 64, pattern:
/^([1-9]|[A-Fa-f0-9]){1,64}$/ },
```

Figure 63.5: The hard-coded regex for the `blockID` parameter of the `/blocks` endpoint is vulnerable to ReDoS.
(`lisk-service/services/gateway/apis/http-version3/methods/blocks.js#26`)

```
2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a2B6a
2B6a2B6a2B6a2B6a2B6a2B6a2B6aAca78abD592AafEFeCAD29aF25dDaa598C2cE52Ba1b74dc6b2F5Db1b
08632
```

Figure 63.6: Input that causes ReDoS for the hard-coded `blockID` pattern

```
const MODULE = /^\b(?:[\w!@$&.]{1,32}|,)+\b$/;
const COMMAND = /^\b(?:[\w!@$&.]{1,32}|,)+\b$/;
```

Figure 63.7: `regex.MODULE` and `regex.COMMAND` in the `blockchain-coordinator` service are vulnerable to ReDoS (but not directly exploitable due to additional validation of the tested parameters).
(`lisk-service/services/blockchain-connector/shared/utils/regex.js#20–21`)

```
zx8dzx8dzx8dzx8dzxzx8dQ$1l88HY&M.{'
```

Figure 63.8: Input that causes ReDoS for `regex.MODULE` and `regex.COMMAND` in the `blockchain-coordinator` service

Exploit Scenario

A user issues the command in figure 63.9 to exploit ReDoS in the `regex.MODULE` expression. Lisk Service consumes excessive CPU while testing the highlighted input, which causes the request to hang indefinitely and block all other requests in the meantime.

```
curl -g
'localhost:9901/api/v3/blocks/assets?module=zx8dzx8dzx8dzx8dzxzx8dQ$1l88HY%26M.{'
```

Figure 63.9: `curl` command that causes ReDoS during parameter validation

Recommendations

Short term, to prevent ReDoS attacks, modify the affected regular expressions to remove the possibility of exponential backtracking.

Long term, ensure that the `js/redos` CodeQL rule is executed in the CI/CD pipeline, and do not allow commits that have findings related to this rule to be pushed to the `development` branch. This will minimize the risk of other vulnerable regular expressions being deployed.

References

- ReDoS, Wikipedia

64. HTTP rate-limiting options are not passed to Gateway container	
Severity: **Medium**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-LISK-64
Target: `lisk-service/docker-compose.yml`	

Description

The `HTTP_RATE_LIMIT_ENABLE`, `HTTP_RATE_LIMIT_CONNECTIONS`, and `HTTP_RATE_LIMIT_WINDOW` environment variables are used to configure HTTP rate limiting for Lisk Service. However, the provided `docker-compose.yml` file does not pass these variables to the Gateway service. As a result, HTTP rate limiting is not enabled in a Docker Compose deployment of Lisk Service, which would be unknown to a user who has correctly set these variables in their environment file.

Exploit Scenario

A user deploys their own instance of Lisk Service using Docker Compose. To prevent denial-of-service attacks, they try to enable HTTP rate limiting by adding the appropriate environment variables to their environment file. When the Gateway container starts, it does not receive these variables, and `HTTP_RATE_LIMIT_ENABLE` falls back to its default setting of `false`, resulting in rate limiting not being enabled.

Recommendations

Short term, add the following lines to the `environment` section of the Gateway service's configuration in `docker-compose.yml`:

```
- HTTP_RATE_LIMIT_ENABLE=${HTTP_RATE_LIMIT_ENABLE}
- HTTP_RATE_LIMIT_CONNECTIONS=${HTTP_RATE_LIMIT_CONNECTIONS}
- HTTP_RATE_LIMIT_WINDOW=${HTTP_RATE_LIMIT_WINDOW}
```

Figure 64.1: Necessary additions to correctly pass HTTP rate-limiting variables to the Gateway container

65. HTTP rate limiter trusts X-Forwarded-For header from client	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-65
Target: `lisk-service/services/gateway/app.js`	

Description

To support reverse proxies, the Gateway service uses the `X-Forwarded-For` header as a key for its HTTP rate-limiting feature. However, it does so by reading the entire value of the header, which may contain untrusted data from the client, instead of securely parsing it to determine the trustworthy client IP address supplied by the reverse proxy (figure 65.1).

```
gatewayConfig.settings.rateLimit = {
      window: (config.rateLimit.window || 10) * 1000,
      limit: config.rateLimit.connectionLimit || 200,
      headers: true,

      key: (req) => req.headers['x-forwarded-for']
             || req.connection.remoteAddress
             || req.socket.remoteAddress
             || req.connection.socket.remoteAddress,
};
```

Figure 65.1: The HTTP rate limiter always reads the entire value of `X-Forwarded-For`, which may contain untrusted data. (`lisk-service/services/gateway/app.js#175–184`)

Unless there is a reverse proxy to overwrite `X-Forwarded-For` (rather than appending to it, as is standard), this use allows clients to bypass any configured rate limit by sending their own `X-Forwarded-For` header.

Exploit Scenario

An instance of Lisk Service is configured to rate limit HTTP requests. An attacker bypasses the rate limit entirely by sending a random value for `X-Forwarded-For` with each request.

Recommendations

Short term, disable reading of the `X-Forwarded-For` header by default. Add a configuration option to enable reading of the header that also specifies the number of reverse proxies between clients and the Gateway service. Once this number is known, determine the correct client IP address in the `X-Forwarded-For` list by counting backwards from the end of the list by the configured number of proxies minus one. For example, with one reverse proxy, the last address in the list is the client's. With two, the second from the last is the client's, and so on.

66. Unhandled exception when filename for transaction history download is a directory

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-LISK-66
Target: `lisk-service/services/export/shared/csvExport.js`, `lisk-service/services/gateway/shared/moleculer-web/methods.js`	

Description

The `/export/download` endpoint, provided by the Export service, allows callers to download the transaction history file referred to by the `filename` parameter. When `filename` is a directory (e.g., a period [`.`]), the file read operation in the `downloadTransactionHistory` method (figure 66.1) throws an `EISDIR: illegal operation on a directory, read` error.

```
const downloadTransactionHistory = async ({ filename }) => {
	const csvResponse = {
		data: {},
		meta: {},
	};

	const isFileExists = await staticFiles.exists(filename);
	if (!isFileExists) throw new NotFoundException(`File ${filename} not found.`);

	csvResponse.data = await staticFiles.read(filename);
```

Figure 66.1: The file read operation throws an exception if `filename` points to a directory. (`lisk-service/services/export/shared/csvExport.js#297–306`)

The `EISDIR` error is then caught and logged in the `httpHandler` method defined for the Moleculer framework. Subsequently, the error is passed to Moleculer's `sendError` method to write it to the client (figure 66.2), which throws a `RangeError` exception because `EISDIR` is not a valid HTTP status code (figure 66.3). This error is not handled, and the Gateway service crashes and restarts as a result.

```
} catch (err) {
	if (this.settings.log4XXResponses || (err && !_.inRange(err.code, 400, 500))) {
		const reqParams = Object
			.fromEntries(new Map(Object.entries(req.$params).filter(([, v]) => v)));
		if (err instanceof ValidationException === false)
```

```
this.logger.error(`<= ${this.coloringStatusCode(err.code)} Request error: ${err.name}: ${err.message} \n${err.stack} \nData: \nRequest params: ${util.inspect(reqParams)} \nRequest body: ${util.inspect(req.body)}`);
            }
            this.sendError(req, res, err);
    }
```

Figure 66.2: The EISDIR error is passed to sendError.
(lisk-service/services/gateway/shared/moleculer-web/methods.js#46–53)

```
RangeError [ERR_HTTP_INVALID_STATUS_CODE]: Invalid status code: EISDIR
    at new NodeError (node:internal/errors:387:5)
    at ServerResponse.writeHead (node:_http_server:314:11)
    at Service.onError (/home/lisk/lisk-service/gateway/app.js:147:10)
    at Service.sendError
(/home/lisk/lisk-service/gateway/node_modules/moleculer-web/src/index.js:859:34)
    at Service.httpHandler
(/home/lisk/lisk-service/gateway/shared/moleculer-web/methods.js:52:10)
    at processTicksAndRejections (node:internal/process/task_queues:96:5) {
  code: 'ERR_HTTP_INVALID_STATUS_CODE'
}
```

Figure 66.3: The Gateway service crashes due to an unhandled RangeError.

Exploit Scenario

An attacker makes an API request to `/export/download` with a `filename` parameter of a period [`.`] by issuing the following command:

```
curl 'localhost:9901/api/v3/export/download?filename=.'
```

Figure 66.4: `curl` command that causes a Gateway crash

Upon processing the parameter, the Gateway service crashes and restarts. The attacker repeatedly runs the command to cause a continuous denial of service to the server.

Recommendations

Short term, have `downloadTransactionHistory` catch and handle errors related to file reads. Add a check to the `httpHandler` method to ensure that the error corresponds to a valid HTTP status code; if it does not, have the code pass a generic code to `sendError`, such as `500 Internal Server Error`.

67. Path traversal in transaction history download	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-67
Target: `lisk-service/services/export/shared/csvExport.js`	

Description

The `/export/download` endpoint allows callers to download the transaction history for a given account. To achieve this, the Export service creates a `FilesystemCache` object around the `lisk-service/export/data/static` directory, where transaction history files reside.

The `downloadTransactionHistory` method then reads the file in this directory that is referenced by the request's `filename` parameter and returns its contents to the caller. However, no sanitization or validation of the filename is performed, and the `read` method for `FilesystemCache` does not prevent traversing out of the defined directory if `filename` is set to, for example, `../../../../etc/passwd`.

Only the contents of files whose names end in .csv will be returned to the caller due to a check in the `onAfterCall` method of the Gateway service. As a result, the impact of this issue is limited to accessing arbitrary CSV files and causing a denial of service by attempting to read large files or files with continuous output, such as `/dev/urandom`.

Exploit Scenario

An attacker makes an API request to `/export/download` with a `filename` parameter of `../../../../../../../../dev/urandom` by issuing the following command:

```
curl
'localhost:9901/api/v3/export/download?filename=../../../../../../../../dev/urandom'
```

Figure 67.1: `curl` command that causes a Gateway crash

The Export service attempts to read `/dev/urandom` and eventually times out, causing the Gateway service to crash and restart (figure 67.2). The attacker repeatedly runs the command to cause a continuous denial of service to the server.

```
2023-06-21T21:07:29.690 WARN [BROKER] Request 'export.transactions.csv' is timed
out.,[object Object]
2023-06-21T21:07:29.693 ERROR [GATEWAY] <= 504 Request error: RequestTimeoutError:
Request is timed out when call 'export.transactions.csv' action on 'da02596db52f-1'
node.
```

```
RequestTimeoutError: Request is timed out when call 'export.transactions.csv' action
on 'da02596db52f-1' node.
    at
/home/lisk/lisk-service/gateway/node_modules/moleculer/src/middlewares/timeout.js:41
:13
    at async Service.callAction
(/home/lisk/lisk-service/gateway/shared/moleculer-web/methods.js:100:16)
    at async
/home/lisk/lisk-service/gateway/node_modules/moleculer-web/src/index.js:469:22
Data:
Request params: { filename: '../../../../../../../../dev/urandom' }
Request body: {}
```

Figure 67.2: The Gateway service crashes due to a timeout when `/dev/urandom` is read.

Recommendations

Short term, have the code ensure that the value of the `filename` parameter conforms to the format `transactions_${address}_${from}_${to}.csv`, as defined in the `getCsvFilenameFromParams` method. Modify the `read` method of the `FilesystemCache` object to prevent path traversal outside the defined root directory. One way to do this is by calling the `path.normalize` method on the path and then ensuring that the resulting string begins with the expected root directory.

References

- Directory traversal, PortSwigger (see the "How to prevent a directory traversal attack" section)

85. Misnamed WebSocket rate-limiting options in Compose file	
Severity: **Medium**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-LISK-85
Target: `lisk-service/docker-compose.yml`	

Description

The `WS_RATE_LIMIT_ENABLE`, `WS_RATE_LIMIT_CONNECTIONS`, and `WS_RATE_LIMIT_DURATION` environment variables are used to configure WebSocket rate limiting for Lisk Service. However, the provided `docker-compose.yml` file passes misnamed variants of these variables to the Gateway service. As a result, WebSocket rate limiting is not enabled in a Docker Compose deployment of Lisk Service, which would be unknown to a user who has correctly set these variables in their environment file.

Exploit Scenario

A user deploys their own instance of Lisk Service using Docker Compose. To prevent denial-of-service attacks, they try to enable WebSocket rate limiting by adding the appropriate environment variables to their environment file. When the Gateway container starts, it does not receive the correct variables, resulting in rate limiting not being enabled.

Recommendations

Short term, replace the incorrect lines in `docker-compose.yml` with the following corrections:

```
- WS_RATE_LIMIT_ENABLE=${WS_RATE_LIMIT_ENABLE}
- WS_RATE_LIMIT_CONNECTIONS=${WS_RATE_LIMIT_CONNECTIONS}
- WS_RATE_LIMIT_DURATION=${WS_RATE_LIMIT_DURATION}
```

Figure 85.1: Necessary replacements to correctly pass WebSocket rate-limiting variables to the Gateway container

Long term, review `docker-compose.yml` to ensure that all variables passed to services align with the configuration reference.

86. Use of hard-coded validation patterns

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-86
Target: Lisk Service API parameter validation	

Description

Several Lisk Service data validation schemas use hard-coded regular expressions, sometimes when an established constant for the required format is available and used elsewhere. For example, the schema for the GET `/blocks` API call hard codes the patterns for its parameters (figure 86.1), despite significant overlap with the schema for the GET `/blocks/assets` call, which uses constants (figure 86.2).

```
blockID: { optional: true, type: 'string', min: 1, max: 64, pattern: /^([1-9]|[A-Fa-f0-9]){1,64}$/ },
height: { optional: true, type: 'string', min: 0, pattern: /([0-9]+|[0-9]+:[0-9]+)/ },
timestamp: { optional: true, type: 'string', min: 1, pattern: /([0-9]+|[0-9]+:[0-9]+)/ },
generatorAddress: { optional: true, type: 'string', min: 38, max: 41, pattern: /^lsk[a-hjkm-z2-9]{38}$/ },
limit: { optional: true, type: 'number', min: 1, max: 100, default: 10, pattern: /^\b((?:[1-9][0-9]?)|100)\b$/ },
offset: { optional: true, type: 'number', min: 0, default: 0, pattern: /^\b([0-9][0-9]*)\b$/ },
```

Figure 86.1: The schema for `/blocks` uses hard-coded regular expressions.
(`lisk-service/services/gateway/apis/http-version3/methods/blocks.js#26–31`)

```
blockID: { optional: true, type: 'string', min: 1, max: 64, pattern: regex.HASH_SHA256 },
height: { optional: true, type: 'string', min: 0, pattern: regex.HEIGHT_RANGE },
timestamp: { optional: true, type: 'string', min: 1, pattern: regex.TIMESTAMP_RANGE },
module: { optional: true, type: 'string', min: 1, pattern: regex.MODULE },
limit: { optional: true, type: 'number', min: 1, max: 100, default: 10 },
offset: { optional: true, type: 'number', min: 0, default: 0 },
```

Figure 86.2: The schema for `/blocks/assets` uses constants for its patterns.
(`lisk-service/services/gateway/apis/http-version3/methods/blockAssets.js#27–32`)

The following Semgrep rule can be used to identify additional instances of hard-coded validation patterns:

```
rules:
  - id: schema_hardcoded_pattern
    message: A fastest-validator schema uses a hardcoded pattern instead of a
constant
    languages: [javascript]
    severity: WARNING
    patterns:
      - pattern-inside: >
          {
              params: {
                  $X: { pattern: /.../ },
              }
          }
```

Figure 86.3: Semgrep rule that detects hard-coded validation patterns

Recommendations

Short term, replace all hard-coded regular expressions with the appropriate existing constants, or define new constants if necessary.

Long term, use the Semgrep rule in figure 86.3 to identify future instances of this issue.

92. Lack of MySQL LIKE escaping in search parameters

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK-92
Target: `lisk-service/framework/src/mysql.js`	

Description

Lisk Service's MySQL interfacing code, which uses the knex library, does not escape search parameters that are used in queries containing the `LIKE` operator (figure 92.1). As a result, users can use MySQL wildcard characters (percent [%] and underscore [_]) in certain parameters, which may not be intended behavior for the API.

```
288    if (params.search) {
289          params.search = Array.isArray(params.search) ? params.search : [params.search];
290
291          params.search.forEach(search => {
292                const { property, pattern, startsWith, endsWith } = search;
293                if (pattern) query.where(`${property}`, 'like', `%${pattern}%`);
294                if (startsWith) query.where(`${property}`, 'like', `${startsWith}%`);
295                if (endsWith) query.where(`${property}`, 'like', `%${endsWith}`);
296          });
297    }
```

Figure 92.1: Search parameters from API calls are passed to MySQL `LIKE` queries without escaping. (`lisk-service/framework/src/mysql.js#288–297`)

This issue is demonstrated by executing the following command, which returns a list of all registered applications with names at least 10 characters long.

```
curl 'http://localhost:9091/api/v3/blockchain/apps?search=__________'
```

Recommendations

Short term, have the code escape all user-controlled parameters used in `LIKE` queries by replacing all occurrences of percent [%] and underscore [_] characters with backsplash percent [\%] and backsplash underscore [_], respectively.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the following issues:

1. **Unused function.** The `executeBlock` function in the `StateMachine` class is exported but never used. This function contains a call to the `executeTransaction` function, but the result is not checked, so if this function was used, invalid transactions could be added to a block.

2. **Inconsistent function name**. The `beforeExecuteBlock` function in the `StateMachine` class should instead be called `beforeTransactionsExecute` to keep the naming consistent with the corresponding module hook and ABI handler.

3. **auth module: unused argument.** The `AuthEndpoint` class unnecessarily receives the `_moduleName` argument. This argument can be safely removed.

4. **auth module: use of hard-coded constant.** In the `initGenesisState` function, the code `storeKey.length !== 20` check uses the hard-coded value `20` where it should instead use the `ADDRESS_LENGTH` constant.

5. **auth module: duplicate implementation of functionality.** The `initGenesisState` and `RegisterMultisignatureCommand.verify` functions perform very similar checks on the creation/modification of multisignature accounts but use different implementations. For example, `initGenesisState` uses the `bufferArrayOrderByLex` function, whereas `RegisterMultisignatureCommand.verify` uses 10 lines of code to check that an array is lexicographically ordered. Checking if a multisignature configuration is correct should be centralized in a single function, which can then be called from both `initGenesisState` and `RegisterMultisignatureCommand.verify`.

6. **token module: duplicate code.** The code `const escrowKey = Buffer.concat([chainID, tokenID])` in the `EscrowSubstore` store should be replaced with the `getKey` method.

7. **token module: duplicate code.** The `deductEscrowAmountWithTerminate` function manually implements the `getOrDefault` function instead of calling the existing method.

8. **token module: duplicate code.** The `isSupported` function uses `const chainID = tokenID.slice(0, CHAIN_ID_LENGTH)` to get the `chainID` instead of using the purpose-built `splitTokenID` function.

9. **token module: dead code.** The `supportToken` function returns early if all tokens are supported but then tries to remove the key that allows the support for all tokens with `this.del(context, ALL_SUPPORTED_TOKENS_KEY)`. The `ALL_SUPPORTED_TOKENS_KEY` key can never exist at this point, which makes this code useless.

10. **token module: unnecessary check.** In the `removeSupportForChain` function, the `if (chainID.equals(this._ownChainID))` check is unnecessary because `_isMainchainOrNative` already performs this check.

11. **token module: poor error handling.** The `token` module throws error messages that are duplicated in multiple parts of the code. For example, errors related to lack of account balance are repeated in `/token/commands/transfer.ts#L57`, `/token/internal_method.ts#L78`, `/token/commands/transfer_cross_chain.ts#L113`, and `/token/commands/transfer_cross_chain.ts#L158`. Instead of hard coding the same error message in multiple locations, consider creating a custom exception that receives the necessary parameters (e.g., sender address, amount) and creates the error message. Then at each location, throw the custom exception.

12. **`validators` module: use of hard-coded constant.** In the `getValidatorKeys` function, the `if (address.length !== 20)` check uses the hard-coded value `20`, where it should instead use the `ADDRESS_LENGTH` constant.

13. **`validators` module: incorrect comparison of parameters.** The `getGeneratorsBetweenTimestamps` method checks that the `endTimestamp` parameter is less than the `startTimestamp` parameter and, if not, throws an error with the message, End `timestamp must be greater than start timestamp`. However, the check would pass if `endTimestamp` and `startTimestamp` were equal, contradicting the content of the error message. The comparison should be rewritten to `if (endTimestamp <= startTimestamp)`.

14. **`lisk-cryptography`: duplicate constants for Lisk32 charset.** The `CHARSET` and `LISK32_CHARSET` constants are equivalent and are both used in the same set of functions for validation and encoding/decoding Lisk32 addresses. The code should be simplified to define and use only one constant for the valid set of characters.

15. **`interoperability` module: use of hard-coded charset string in error messages.** Several error messages in the `interoperability` module and its unit

tests hard code the character set for a valid chain name instead of using the `validNameCharset` constant. These instances are listed below:

- `C-15-1`
- `C-15-2`
- `C-15-3`
- `C-15-4`
- `C-15-5`
- `C-15-6`

16. **`interoperability` module: duplicate implementation of functionality.** The `interoperability` module calculates a CCM ID by encoding the CCM and hashing the result. The code hard codes this in multiple places instead of implementing it in a single, centralized function. This code is repeated in `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/submit_mainchain_cross_chain_update.ts#L138`, `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/recover_message.ts#L410`, and `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/recover_message.ts#L188`. Instead, the `getEncodedCCMAndID` function could be used. Alternatively, another function that calculates the ID from the encoded CCM could also be created.

17. **`interoperability` module: incorrect error message.** In the `forward` method, an error message says `Execute beforeCrossChainCommandExecute`, but the code executes `beforeCrossChainMessageForwarding` instead. This error occurs in two locations: `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/submit_mainchain_cross_chain_update.ts#L247-L250`, and `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/submit_mainchain_cross_chain_update.ts#L233-L240`.

18. **`interoperability` module: lack of schema property length.** The `ChainAccountStore` schema name property is missing the length validation described in the corresponding LIP.

19. **`interoperability` module: unnecessarily complex code.** The `ChainAccountStore` class's `getAllAccounts` function could be simplified. Currently, the code uses `this.get(context, chainAccount.key)` to get the value of each `chainAccount`, but it could simply use `chainAccount.value`.

20. `interoperability` **module: lack of schema property length.** The `OwnChainAccountStore` schema name property is missing the length validation described in the corresponding LIP.

21. `interoperability` **module: misspelling.** LIP-0045 misspells `mainchain` as `manchain`.

22. `interoperability` **module: lack of schema validation.** The certificate schema is decoded but not validated in several locations. In most cases but not all, the LIP specifies that this validation should be present. Both the code and the LIP should be updated to reflect each other. The following are the problematic locations:

    - `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/submit_mainchain_cross_chain_update.ts#L270`
    - `lisk-sdk/framework/src/modules/interoperability/utils.ts#L268`
    - `lisk-sdk/framework/src/modules/interoperability/utils.ts#L216`
    - `lisk-sdk/framework/src/modules/interoperability/utils.ts#L162`
    - `lisk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L636`
    - `lisk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L407`
    - `isk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L373`
    - `lisk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L361`
    - `lisk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L272`

23. `interoperability` **module: repeated code.** The `verifyLivenessConditionForRegisteredChains` function is reimplemented in two locations: `lisk-sdk/framework/src/modules/interoperability/utils.ts#L264-L275` and `lisk-sdk/framework/src/modules/interoperability/mainchain/commands/submit_mainchain_cross_chain_update.ts#L267-L278`.

24. **`interoperability` module: error message typo.** Two error messages in the `interoperability` module contain the word `non-mepty` instead of `non-empty`:

    - `lisk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L595`
    - `lisk-sdk/framework/src/modules/interoperability/base_interoperability_internal_methods.ts#L600`

25. **`interoperability` module: repeated code.** The `verify` methods of the `MainchainCrossChainUpdate` and `SidechainCrossChainUpdate` commands have repeated code that could be centralized in a single function. This is what already happens for the `execute` function of the same commands with the `executeCommon` function.

26. **`interoperability` module: unused parameter.** In the `SidechainCrossChainUpdate` command, the `execute` method's `ccmContext` variable contains an additional ccu property that is not specified in the LIP. This value seems unused in the `apply` function, so it could safely be removed.

27. **Certificate generation: use of hard-coded constant.** To align with LIP-0061 and best practices for the use of constants, the following buffer length checks against `0` should instead compare the length of the buffer against the length of the `EMPTY_BUFFER` constant:

    - `lisk-sdk/framework/src/engine/consensus/certificate_generation/commit_pool.ts#L190-L191`
    - `lisk-sdk/framework/src/engine/consensus/certificate_generation/commit_pool.ts#L198-L199`

28. **`interoperability` module: missing check.** The `getMessageFeeTokenID` method is missing the own-chain check described in the LIP. This check is implemented in other functions such as `getMinReturnFeePerByte`. Calculation of the `updatedChainID` variable should be centralized in a single function to avoid these discrepancies.

29. **`interoperability` module: incorrect comment.** The comment `The commands fails if the sidechain is already terminated on this chain` is incorrect. Commands fail only if the sidechain is terminated and initialized on this chain. The whole purpose of the command is to initialize the state of the terminated chain.

30. **`interoperability` module: lack of schema validation.** The `ChainAccount` provided by a user in the `InitializeStateRecoveryCommand` command is decoded but not validated. The inclusion proof prevents rogue values from being

used (although a malicious chain could create invalid accounts). Additionally, passing the empty string value to trigger a noninclusion proof is impossible.

31. **`interoperability` module: lack of schema validation.** The `ChannelData` provided by the user in the `InitializeMessageRecoveryCommand` command is decoded but not validated. The inclusion proof prevents rogue values from being used (although a malicious chain could create invalid accounts). Additionally, passing the empty string value to trigger a noninclusion proof is possible (see TOB-LISK-47), but the decoding would fail in the `execute` method.

32. **`interoperability` module: unnecessarily complex code.** The implementation of the `updatedChainID` variable in the `getMinReturnFeePerByte` method is hard to read due to the use of inline `if`s and ternary operators. The pseudocode in the LIP is easier to read. Modify the code to be more readable.

33. **`interoperability` module: missing checks.** The `isChainNameAvailable` endpoint is missing the size checks described in LIP-0045.

34. **`interoperability` module: poor validation.** The `isChainNameAvailable` endpoint performs poor validation of its parameter's schema. While most other functions use the `lisk-validator` package, this function uses the `typeof` keyword to directly validate the types of its parameters.

35. **`lisk-mobile`: use of a deprecated Clipboard feature**. Documentation recommends switching to community packages.

36. **`lisk-desktop`: unused parameters.** The `state.origin` attribute and the `state.timeoutObj` attribute of the Multistep component are written to but never read.

37. **`lisk-desktop`: lack of string internationalization.** Most strings in `lisk-desktop` are internationalized to facilitate the application's use in different languages. However, 30 different strings are hard coded and lack internationalization (e.g., the string `Create one now` in the `AddAccountOptions component`). We recommend running the `i18next.jsx-not-internationalized` Semgrep rule to find and fix these issues.

38. **`lisk-desktop`: incorrect JDocs.** `lisk-desktop` uses JDocs to document functions; however, several are malformed, are missing, or include unknown parameters. Use the following CodeQL rules to find and fix these problems:

    - `js/jsdoc/malformed-param-tag`: finds parameters missing a description
    - `js/jsdoc/missing-parameter`: finds undocumented parameters

- `js/jsdoc/unknown-parameter`: finds documented parameters that are not function arguments

39. **`lisk-mobile`: the iOS `Info.plist` file contains deprecated keys.** For example, the `NSLocationAlwaysUsageDescription` property works only for iOS before version 11, and the `NSLocationAlwaysAndWhenInUseUsageDescription` property should be used for newer versions.

40. **`lisk-mobile`: the `getRecoveryPhraseFromKeyChain` function name is misleading.** The function retrieves a password, not the recovery phrase, from the keychain. The recovery phrase is stored in encrypted form in a file on a disk.

41. **`lisk-mobile`: lack of support for Privacy Dashboard in Android version of `lisk-mobile`.** The `lisk-mobile` application should provide a rationale for why it is using specific features.

42. **`lisk-mobile`: Android's `android:usesCleartextTraffic` option is set to `true`.** Setting this option to `false` will prevent accidental use of plaintext HTTP traffic. The default value for API versions 27 and below is `true`. Also consider using the `cleartextTrafficPermitted` setting.

43. **`lisk-mobile`: Android's `ProviderInstaller` is not used.** The `lisk-mobile` application should explicitly check whether it is running on a device that has an up-to-date Android security provider.

44. **`lisk-mobile`: Android's `android:dataExtractionRules` rule is not used to control what data is transferred between devices.** This setting should be used in addition to the `android:allowBackup` setting, which provides less control over data transfers and backups.

45. **`lisk-mobile`: Android SDK version should be updated to version 31 (Android 12) to prevent TapJacking attacks.** If this is not feasible, set the `View.setFilterTouchesWhenObscured(true)` flag as described in the Mitigations section of the Android documentation.

46. **`lisk-mobile`: device does not enable Verify Apps during `lisk-mobile` application startup.** Enabling this will provide `lisk-mobile` with a stronger level of protection from malicious third-party applications.

47. **`lisk-mobile`: Android ID (the `getUniqueId` method) is used as a username for keychain storage.** Using the Android ID may raise concerns about users' privacy, and it forces the `READ_PHONE_STATE` permission to be used, which may be alarming for users. Use an application-wide constant instead, possibly concatenated with a random string.

48. **`lisk-mobile`: inconsistent validation of the "Amount" field in "Send token" functionality of `lisk-mobile`.** If no amount is provided at all, then it is always accepted. However, providing `0` as the amount is not acceptable if a user has no balance.

49. **`lisk-desktop`: incorrect `setState` callbacks**. Several `setState` callbacks use the `this.props` or `this.state` variables, which is incorrect, as explained in React's documentation. Using the `js/react/inconsistent-state-update` CodeQL rule, we found the following problematic instances:

    - `lisk-desktop/src/modules/bookmark/components/BookmarksList/BookmarksList.js#L89-L92`
    - `lisk-desktop/src/modules/wallet/components/passphraseRenderer/index.js#L126-L133`
    - `lisk-desktop/src/modules/wallet/components/searchBarWallets/passphraseRenderer/index.js#L126-L133`
    - `lisk-desktop/src/theme/tabs/tabsContainer/tabsContainer.js#L49-L54`
    - `lisk-desktop/src/theme/Tooltip/tooltip.js#L32-L34`
    - `lisk-desktop/src/utils/withData.js#L113-L120`
    - `lisk-desktop/src/utils/withFilters.js#L39-L44`
    - `lisk-desktop/src/utils/withFilters.js#L65-L76`
    - `lisk-desktop/src/utils/withLocalSort.js#L26-L28`

50. **`lisk-desktop`: nonexistent variable imported**. The `useSendTransaction` hook imports the `API_METHOD` variable from the `src/const/config` file; however, this variable does not exist in the config file.

51. **`lisk-desktop`: incorrect props passed**. The CustomRoute component is called in the `MainRouter.js` file with a `key` property and a `route` property that `CustomRoute` does not use. Furthermore, the `t` parameter is not passed in. This likely happens in many other components, so we recommend creating a CodeQL rule (or similar) to find other cases.

52. **`lisk-desktop`: unused route property**. The `isSigninFlow` property of routes is never used.

53. **`lisk-desktop`: unrouted modal**. The `setPassword` modal is defined in the `route.js` file, but there is no mapping to its component in the `routesMap.js` file.

54. **`lisk-desktop`: modal screen leads to crash**. Navigating to the `selectNode` modal (e.g., http://localhost:8080/#/?modal=selectNode) causes the app to crash.

55. **`lisk-db`: the `update` and `new` methods are ambiguous.** These methods use a hash of a zero-length string as a tree root if either an empty buffer or a hash of a zero-length string is provided as the first argument.

```
const root = await eventSMT.update(Buffer.alloc(0), data);
```

Figure C.1: Example call with empty buffer (`lisk-sdk/framework/src/engine/endpoint/chain.ts#215`)

```
const eventRoot = await smt.update(EMPTY_HASH, keypairs);
```

Figure C.2: Example call with hash of empty buffer (`lisk-sdk/framework/src/engine/consensus/consensus.ts#839`)

56. **`lisk-sdk`: unused parameter in `token` module configuration schema**. The schema used to validate the `token` module's configuration contains a parameter called `supportedTokenIDs`, which is not present in the `ModuleConfig` options defined by the module.

D. LIP-to-Implementation Discrepancies

This appendix describes all the discrepancies of informational severity between LIPs and their corresponding implementations that we found during the audit. We recommend that for each, either the code is fixed or the LIP is updated so that both match.

LIP-0039

There are differences between LIP-0039 and the sparse Merkle tree implementation.

The first discrepancy is related to the `filterQueries` function defined in LIP-0039, shown in figure D.1. The function throws an error when two query keys with a common prefix are detected. However, `lisk-db`'s implementation simply filters out queries with the same prefix, as shown in figures D.2 and D.3.

```
// This function filters the array of queries by keeping only those
// with a different key prefix, i.e., by removing queries that have
// merged together
function filterQueries(queries):
    for each query q in queries:
        let h=length of q.binaryBitmap
        let binaryKey=binaryExpansion(q.key)
        let q.keyPrefix be the first h digits of binaryKey

        if there exists another p in queries s.t. p.keyPrefix == q.keyPrefix:
            if p.hash != q.hash:
                // the two merging queries have mismatching hashes
                throw error
            // filter queries by keeping only unique values of q.keyPrefix
            remove p from queries

    return queries
```

Figure D.1: The `filterQueries` function from LIP-0039

```
pub fn prepare_queries_with_proof_map(
    proof: &Proof,
) -> HashMap<Vec<bool>, QueryProofWithProof> {
    let mut queries_with_proof: HashMap<Vec<bool>, QueryProofWithProof> =
HashMap::new();
    for query in &proof.queries {
        let binary_bitmap =
utils::strip_left_false(&utils::bytes_to_bools(&query.bitmap));
        let binary_path =
utils::bytes_to_bools(&query.pair.0)[..binary_bitmap.len()].to_vec();
```

```
        queries_with_proof.insert(
            binary_path,
            QueryProofWithProof::new_with_pair(
                Arc::clone(&query.pair),
                &binary_bitmap,
                &[],
                &[],
            ),
        );
    }

    queries_with_proof
}
```

Figure D.2: Implementation of the `filterQueries` method (`lisk-db/src/sparse_merkle_tree/smt.rs#1363–1383`)

```
if !utils::array_equal_bool(&q.binary_path(), &original.binary_path()) {
    queries.insert(index as usize, q);
}
```

Figure D.3: The second, optimized implementation of the `filterQueries` method (`lisk-db/src/sparse_merkle_tree/smt.rs#362–364`)

The second discrepancy is related to the `areSiblingQueries` method, shown in figure D.4. This method returns `true` only if the first argument is the left child and the second argument is the right child of a common parent. However, `lisk-db`'s implementation, shown in figure D.5, does not differentiate between left and right children.

```
// This function checks whether two queries correspond to nodes that are
// children of the same branch node, with q1 and q2 the left and right
// child respectively
function areSiblingQueries(q1, q2):
    if length of q1.binaryBitmap != length of q2.binaryBitmap:
        return false
    let h=length of q1.binaryBitmap
    let binaryKey1=binaryExpansion(q1.key)
    let binaryKey2=binaryExpansion(q2.key)
    let keyPrefix1 be the first (h-1) digits of binaryKey1
    let keyPrefix2 be the first (h-1) digits of binaryKey2
    if keyPrefix1 != keyPrefix2:
        return false
    let d1 be the digit at position h of binaryKey1
    let d2 be the digit at position h of binaryKey2
    return d1==0 and d2==1
```

Figure D.4: The `areSiblingQueries` method from LIP-0039

```
fn is_sibling_of(&self, query: &QueryProofWithProof) -> bool {
    [skipped]

    if !self.binary_key()[self.height() - 1] && query.binary_key()[self.height() -
1] {
        return true;
    }

    if self.binary_key()[self.height() - 1] && !query.binary_key()[self.height() -
1] {
        return true;
    }
    false
}
```

Figure D.5: The part of the `areSiblingQueries` implementation that deviates from the specification (`lisk-db/src/sparse_merkle_tree/smt.rs#582–603`)

LIP-0041

There are differences between LIP-0041 and the implementation of the `auth` module.

The `genesisAuthStoreSchema` schema has different names for its items (`address` and `authAccount` in LIP-0041 versus `storeKey` and `storeValue` in the implementation). The implementation also lacks length checks on several fields that are defined in the LIP schema, which means genesis data may be malformed (e.g., the size of the mandatory and optional keys may be malformed).

The `getAuthAccount` method implementation differs from LIP-0041. The LIP says the implementation should raise an exception when no account exists, whereas the implementation instead returns a default account when no account exists.

The `getAuthAccount` endpoint implementation differs from LIP-0041. The LIP says it is identical to the `getAuthAccount` method; however, it receives a `lisk32address` instead of an account address.

The endpoints `getMultiSigRegMsgSchema`, `sortMultisignatureGroup`, and `getMultiSigRegMsgTag` are not specified in the LIP but are implemented.

The `verifyNonce` function implementation differs from LIP-0041. The LIP makes no mention of the `PENDING` state that the `verifyNonce` function may return. Also, the consensus code handles `PENDING` the same way it handles `FAIL`, making it redundant in the current state of the code.

Even though these differences did not lead to security issues, some changes, such the addition of the `PENDING` state, make it harder to audit the code's correctness.

LIP-0051

There are differences between LIP-0051 and the implementation of the `token` module.

LIP-0051 states that a sidechain must support its native tokens but gives this as an example of a chain setup: "A chain with a specific use case and no native token could only support the LSK token." In this example, the chain does not support its native tokens.

LIP-0051 states: "Lastly, note that modifying the list of supported tokens would result in a fork of the chain. For this reason, the default behavior for Lisk sidechains would be to support all tokens." However, there are methods that allow other modules to modify the list of supported tokens.

LIP-0051 states that a module name in the user substore should match the following regex pattern: `^[a-zA-Z0-9]*$`. However, the implementation's schema does not perform this check.

The `transferCrossChainInternal` function described in LIP-0051 does not exist in the implementation.

Several functions specified in the LIP have different names in the implementation. The `getTokenIDLSK` function is called `getMainchainTokenID` in the code. The `userSubstoreExists` function is called `userAccountExists` in the code.

LIP-0051 does not list the module's endpoints.

LIP-0051 contains several "TBD" strings.

LIP-0051 states in multiple places that addresses should be obtained with `senderAddress = SHA256(trs.senderPublicKey)[:ADDRESS_LENGTH] # Derive sender address from trs.senderPublicKey`. This adds complexity for no benefit; in the code, this is just a call to `transaction.sender`.

LIP-0051's `getTotalSupply` function receives a `tokenID`, but the implementation does not. Instead, it returns the total supply for all tokens.

The `TransferCrossChainCommand` schema has the `recipientAddress` property with the `lisk32` format, but the LIP uses the `bytes` format with a max length.

The cross-chain token transfer schema has the `senderAddress` and `recipientAddress` properties with the `lisk32` format, but the LIP uses the `bytes` format with a max length.

LIP-0049

LIP-0049 states that the channel terminated message includes the `verify` method, but this method does not exist in the code. The method is unnecessary because it receives no parameter, so an empty version should be added to the code, or it should be removed from the LIP.

LIP-0043

LIP-0043's pseudocode for the `verify` method of the sidechain registration command iterates over the provided sidechain validators, summing their `bftWeight` properties. If at any point in the loop the sum exceeds the `MAX_UINT64` constant, the function raises an exception (figure D.6).

```
totalWeight = 0
for validator in trsParams.sidechainValidators:
    # The bftWeight property of each element is a positive integer.
    if validator.bftWeight == 0:
        raise Exception("Invalid bftWeight property.")
    totalWeight += validator.bftWeight
    # Total BFT weight has to be less than or equal to MAX_UINT64.
    if totalWeight > MAX_UINT64:
        raise Exception("Total BFT weight exceeds maximum value.")
```

Figure D.6: The pseudocode checks if `totalWeight` exceeds `MAX_UINT64` on every iteration of the loop.

However, the `lisk-sdk` implementation of this function verifies that the sum does not exceed `MAX_UINT64` only after the loop has terminated (figure D.7). The code should be modified to perform this check within the loop, in accordance with the LIP.

```
	let totalBftWeight = BigInt(0);
	for (let i = 0; i < sidechainValidators.length; i += 1) {
		const currentValidator = sidechainValidators[i];

		// The blsKeys must be lexicographically ordered and unique within the array.
		if (
			sidechainValidators[i + 1] &&
			currentValidator.blsKey.compare(sidechainValidators[i + 1].blsKey) > -1
		) {
			return {
				status: VerifyStatus.FAIL,
				error: new Error('Validators blsKeys must be unique and lexicographically ordered'),
			};
		}

		if (currentValidator.bftWeight <= BigInt(0)) {
			return {
				status: VerifyStatus.FAIL,
				error: new Error('Validator bft weight must be greater than 0'),
```

```
            };
        }

        totalBftWeight += currentValidator.bftWeight;
    }

    if (totalBftWeight > MAX_UINT64) {
        return {
            status: VerifyStatus.FAIL,
            error: new Error(`Validator bft weight must not exceed
${MAX_UINT64}`),
        };
    }
```

Figure D.7: The implementation checks if `totalBftWeight` exceeds `MAX_UINT64` after the loop ends.
(`lisk-sdk/framework/src/modules/interoperability/mainchain/commands/register_sidechain.ts#128–158`)

LIP-0053

The `MainchainCrossChainUpdate` and `SidechainCrossChainUpdate` commands defined in LIP-0053 share common functionality. However, in the LIP, instead of making these differences clear by describing a common function with the shared functionality, the LIP copy-pastes the shared pseudocode in both commands. This makes the differences between mainchain and sidechain commands harder to understand. The code has the common functionality centralized in `*common` functions. The LIP should reflect that.

The `MainchainCrossChainUpdate` command's `verify` function pseudocode in the LIP has the code `len(ccu.params.inboxUpdate) > 0`. Instead, the pseudocode should be `ccu.params.inboxUpdate is empty`, as in other parts of the pseudocode.

In the `SidechainCrossChainUpdate` command, the `execute` method's `ccmContext` variable does not contain the `ccmID` topic by default as the LIP specifies. The `MainchainCrossChainUpdate` does this correctly.

LIP-0045

The LIP's Message Recovery Command description misspells identifying as "identifyng."

The `getChainValidators` method is described in the LIP but does not exist as a method. This method exists only as an endpoint in the `interoperability` module.

The LIP describes only the `isChainIDAvailable` and `isChainNameAvailable` endpoints, but many others exist.

E. Dynamic Analysis Configuration

This appendix describes the setup of the automated dynamic analysis tools and test harnesses used during this audit.

The Purpose of Automated Dynamic Analysis

In most software, unit and integration tests are typically the extent of the testing performed. This type of testing detects the presence of functionality, allowing developers to ensure that the given system adheres to the expected specification. However, these methods of testing do not account for other potential behaviors that an implementation may exhibit.

Fuzzing and property-based testing complement both unit and integration testing by identifying deviations in the expected behavior of a system component. These types of tests generate test cases and provide them to the given component as input. The tests then run the components and observe their execution for deviations.

The primary difference between fuzzing and property testing is the method of generating inputs and observing behavior. Fuzzing typically attempts to provide random or randomly mutated inputs in an attempt to identify edge cases in entire components. Property testing provides inputs sequentially or randomly within a given format, checking to ensure a specific system property holds upon each execution.

By developing fuzzing and property-based testing alongside the traditional set of unit and integration tests, edge cases and unintended behaviors can be pruned during the development process, which will likely improve the overall security posture and stability of a system.

Configuring the Fuzzer

To fuzz test JavaScript and TypeScript codebases, we recommend using the `jazzer.js` tool. Recently released version 1.5.0 enables easy integration with TypeScript's Jest testing framework. Because this version was released during the audit, we present some of the harnesses in the old format without the Jest integration.

The `jazzer.js` documentation provides guidance for the installation and configuration processes.

An important configuration option—or a Jazzer's runtime flag—is the `--instrumentation_includes` flag. It can be used to adjust what parts of the codebase and dependencies should be instrumented. This flag greatly impacts effectiveness and speed of fuzzing, so the best values for the flag should be experimentally determined.

Creating Harnesses

To start with fuzzing harness development, we need a utility function to differentiate legitimate errors—such as exceptions thrown by validation methods—from errors that we are interested in finding and that indicate a bug in the code under test.

```
const knownErrors = [
       "Invalid ",
       "Unknown format ",
       "Value yields unsupported wireType",
       "Compiled Schema is corrupted",
       "Message does not contain a property for",
       "Value out of range ",
       "Terminating bit not found"
];

function isKnownError(msg: string) {
       for (const ke of knownErrors)     {
              if (msg.startsWith(ke))
                     return true;
       }
       return false;
}
```

Figure E.1: A function to skip safe errors

A better mechanism for handling safe (expected) errors should be developed. For example, the Lisk codebase should have a single `root` error type specific to `lisk-sdk`. Then specific modules should inherit from that error type to implement per-module error types. With this design, filtering out safe exceptions would be easier and more efficient.

Transaction round-trip property

A basic fuzzing harness for testing encoding and decoding methods for a Lisk transaction is shown in figure E.2. The fuzzing harness decodes `bytes` as a transaction, then (optionally) validates it to skip testing invalid transactions. After that, the transaction is encoded to `bytes` and decoded again. Finally, the test checks that the initial transaction `bytes` are equal to the `bytes` after the round trip and throws an error if they are not.

```
it.fuzz('fuzz', (data: Buffer) => {
       try {
              const tx = Transaction.fromBytes(data);

              // comment lines below to be perform less strict fuzzing
              try {
                     tx.validate();
              } catch (e) {
                     return;
              }
              // stop comment
```

```
            const data2 =  tx.getBytes();
            const tx2 = Transaction.fromBytes(data2);
            if (tx !== tx2) {
                  throw new Error(Found tx round robin bug - tx);
            }
            if (Buffer.compare(data, data2) !== 0) {
                  throw new Error(Found tx round robin bug - bytes);
            }
      } catch (e) {
            if (!isKnownError((e as Error).message)) {
                  throw e;
            }
      }
});
```

Figure E.2: Fuzzing harness for testing round-trip transaction decoding and encoding

Codecs correctness

The fuzzing harness shown in figure E.3 tests correctness of the simplest `lisk-codec` methods. These tests parse input data as a variable with a type under test, then write the variable as `bytes`. If the relevant part of the input data is not equal to the `bytes` after write, the harness reports an error.

Figure E.3 presents dozens of fuzzing harnesses that must be configured to run separately, as the Jazzer tool supports running only a single fuzzing harness at a time. Moreover, the `isKnownError` error may need to be adjusted to account for different safe exceptions.

```
function testReadWrite(data: Buffer, name: String, read: (buffer: Buffer, offset:
number) => [any, number], write: (value: any) => Buffer) {
      try {
            const [n, size] = read(data, 0);
            const data2 = write(n);
            if (Buffer.compare(data.slice(0, size), data2) !== 0) {
                  throw new Error(`Found read/write bug - ${name}`);
            }
      } catch (e) {
            if (!isKnownError((e as Error).message)) {
                  throw e;
            }
      }
}

export function fuzz(data: Buffer) {
      testReadWrite(data, "uint32", readUInt32, writeUInt32);
}
export function fuzz2(data: Buffer) {
      testReadWrite(data, "sint32", readSInt32, writeSInt32);
}
export function fuzz3(data: Buffer) {
      testReadWrite(data, "uint64", readUInt64, writeUInt64);
}
```

```
export function fuzz4(data: Buffer) {
	testReadWrite(data, "sint64", readSInt64, writeSInt64);
}
export function fuzz5(data: Buffer) {
	testReadWrite(data, "string", readString, writeString);
}
export function fuzz6(data: Buffer) {
	testReadWrite(data, "bytes", readBytes, writeBytes);
}
```

Figure E.3: Fuzzing `lisk-codec` basic methods

A more complex example is shown in figure E.4, where an object's read and write methods are tested. First, the `beforeAll` method is used to compile an example schema before the fuzzing starts. Then the harness checks whether an object—if successfully decoded from `bytes`—can be correctly encoded again to `bytes`.

Two possible improvements could be made to the harness:

- The example schema used in the code in figure E.4 could be expanded to include a more comprehensive set of supported types. Ideally, the schema used in fuzzing should include all supported types, including subtypes and validation requirements.
- Many fuzzing resources are wasted on decoding invalid objects. A better approach to writing the fuzzing harness would be to use the `FuzzedDataProvider` class to construct a valid object for the example schema, encode it to `bytes`, decode the `bytes` to a new object, and then test the interesting properties on the two objects, instead of the `bytes`.

```
beforeAll(() => {
	const exampleSchema = {
		"$id": "/exampleSchema",
		"type": "object",
		"properties": {
			"moduleID": {
				"dataType": "uint32",
				"fieldNumber": 1
			},
			"assetID": {
				"dataType": "bytes",
				"fieldNumber": 2
			}
		},
		"required": [
			"moduleID",
			"assetID"
		]
	};
	codec.addSchema(exampleSchema);
});
```

```
export function fuzz(data: Buffer) {
	const schemaCompiled = (codec as any)._compileSchemas['/exampleSchema'];
	try {
		const [obj, size] = readObject(data, 0, schemaCompiled, data.length);
		if (size != data.length) {
			throw new Error(`Found read/write bug - object - sizes`);
		}
		const [chunks, chunks_total_size] = writeObject(schemaCompiled, obj,
[]);
		if (chunks.reduce((acc, val) => acc + val.length, 0) !=
chunks_total_size) {
			throw new Error(`Found read/write bug - object - total sizes`);
		}
		if (Buffer.compare(data, Buffer.concat(chunks)) !== 0) {
			throw new Error(`Found read/write bug - object - data`);
		}
	} catch (e) {
		if (!isKnownError((e as Error).message)) {
			throw e;
		}
	}
}
```

Figure E.4: A harness for testing object encoding and decoding

Expanding Fuzzing Coverage

The following are ideas of fuzzing harnesses that can be constructed to dynamically test various invariants that the `lisk-sdk` codebase depends on. The list is a work in progress.

- The `fromJSON`, `toJSON`, `fromBytes`, and `getBytes` methods of various objects could be tested with a round-trip type of fuzzer. Testing should involve a diverse set of schemas.

- Transactions' and blocks' signing mechanisms should be validated to be complete—that is, signatures should cover all bytes of a transaction or a block. This can be fuzz tested by constructing a valid transaction or block object, computing a valid signature for it, and then modifying the object and checking whether the signature is still valid. If it is, some parts of the object were not signed. The multisignature feature could also be tested by mutating both transaction data and a set of valid and invalid signatures.

- Any method performing arithmetic computation or validation could be fuzz tested against integer overflows and rounding errors. In particular, methods that operate on numbers that are not `BigInts` and that involve binary operations (transforming numbers to 32-bit integers) should be tested.

- The peer-to-peer API should be tested against data coming from multiple malicious peers. Fuzz testing should look for both crashes (i.e., unexpected and unhandled exceptions) and denial-of-service (DoS) conditions (e.g., extensive memory use, extensive disk space occupation, long response times).
- Schema compilation should be tested for DoS resistance (e.g., infinite recursion).
- All stages of command (or transaction or block) processing should be tested to ensure they are executed in a timely manner. Tests should be done with the maximum expected fees to verify bound computations for the worst case scenario.
- The `token` module could be tested with common properties like the ones listed below. Testing should include states and methods introduced by the `interoperability` module.
    - The balance of one user must be less than or equal to the total supply.
    - The sum of balances of all users must equal the total supply.
    - A user cannot transfer more than his balance.
    - The total supply does not increase or decrease without minting or burning.
- The `interoperability` module can be fuzz tested to ensure that the following properties hold (this list is non-exhaustive):
    - Users' locked funds are never moved until they request to unlock them.
    - On a destination chain, funds are not minted unless an equivalent amount of funds is securely locked on the source chain.
    - Minting and movement of funds on the destination chain should not affect whether funds are locked on the source chain.
    - Recovery messages cannot be replayed.
- The sparse Merkle tree implementation could be fuzz tested against the following properties.
    - If an item was added to a tree, an inclusion proof for that item is always `true`. The property should hold even after arbitrary other items are added to and removed from the tree.
    - If an item was added to a tree, a noninclusion proof for that item cannot be constructed.

- If an item was not added to a tree, or added and later removed, a noninclusion proof for that item is always `true`.
- A new item can always be added to an arbitrary tree in a short time.
- The state can be snapshotted and reverted correctly.

All tests should be performed for both in-memory and RocksDB back ends. Moreover, the two back ends can be cross-tested against each other—that is, fuzzing should aim to find any differences in behavior between the two.

These fuzz tests may be implemented in both Rust and TypeScript. While we recommend implementing both, the TypeScript fuzzing has the potential to reveal issues in the foreign function interface (FFI).

- Differential fuzzing for the sparse Merkle tree implementation would be profitable. For example, Penumbra's Jellyfish Merkle Tree implementation could be used as a reference implementation.
- We recommend going over each LIP and doing an exercise to extract invariants and properties that can be tested with fuzzing.

F. Static Analysis Tool Configuration

As part of this assessment, we performed automated static testing using Semgrep and CodeQL.

Semgrep

We performed static analysis on the Lisk SDK, Lisk Desktop, Lisk Mobile, and Lisk Service components using Semgrep. We used several TypeScript and JavaScript rules, including those from publicly available rulesets, as well as those we developed specifically for Lisk. For all targets, we used the following public rulesets:

- `p/javascript`
- `p/r2c`
- `p/r2c-security-audit`
- `p/r2c-best-practices`
- `p/nodejs`
- `p/nodejsscan`
- `p/react`

We also used the following custom rules on specific targets (see appendix G):

- `error_event_is_not_revert`
- `get_modify_no_set_on_stores`
- `module_registration_of_correct_class`
- `module_stores_same_index`
- `schema_min_max_items_without_array` (TOB-LISK-34)
- `schema_property_element_without_field_number`
- `schema_with_duplicate_field_number`
- `schema_with_field_not_required` (TOB-LISK-28)
- `verify_without_schema_verify` (TOB-LISK-30)
- `schema_with_required_that_is_not_a_property` (TOB-LISK-26)

- `schema_int_format_with_integer_type` (TOB-LISK-52)
- `schema_hardcoded_pattern` (TOB-LISK-86)

CodeQL

We used CodeQL to analyze the Lisk codebases. We used our private `tob-javascript-all` query suite, which includes public JavaScript queries and some private queries. Figure F.1 shows the commands used to create the CodeQL database and run the queries.

```
# Create the javascript database
codeql database create codeql.db --language=javascript

# Run all javascript queries
codeql database analyze codeql.db --format=sarif-latest --output=codeql_res.sarif --
tob-javascript-all.qls
```

Figure F.1: Commands used to run CodeQL

G. Custom Semgrep Rules

The following appendix lists all the Semgrep rules that were created specifically for the Lisk codebase. We recommend running these rules in the CI/CD pipeline. At the end of the appendix, we provide ideas for more rules.

Custom Rules

The following rule detects problems in an event class:

```
rules:
  - id: error_event_is_not_revert
    message: An event class has an error function that adds an event that is not
persisted.
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            class $CLASS extends BaseEvent {
              ...
            }
        - pattern: >
            error(...) {
                ...
              }
        - pattern-not: >
            error(...) {
                ...
                this.add(..., true);
                ...
              }
```

Figure G.1: The `error_event_is_not_revert` Semgrep rule

The following rule detects store elements that are read and modified but never saved:

```
rules:
  - id: get_modify_no_set_on_stores
    message: $A is read, modified, but never saved to storage
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            $FUNC(...) {
                ...
                $STORE = this.stores.get($STORE_TYPE);
                ...
                $A = await $STORE.get(...);
                ...
                $A.$FIELD = $VALUE;
```

```
                ...
            }
        - pattern-not-inside: >
            $FUNC(...) {
                ...
                $STORE = this.stores.get($STORE_TYPE);
                ...
                $A = await $STORE.get(...);
                ...
                $A.$FIELD = $VALUE;
                ...
                await $STORE.set(...);
                ...
            }
        - pattern-not-inside: >
            $FUNC(...) {
                ...
                $STORE = this.stores.get($STORE_TYPE);
                ...
                $A = await $STORE.get(...);
                ...
                $A.$FIELD = $VALUE;
                ...
                await $STORE.save(...);
                ...
            }
        - pattern: >
            $A.$FIELD = $VALUE;
```

Figure G.2: The `get_modify_no_set_on_stores` Semgrep rule

The following two rules detect the incorrect registration of classes:

```
rules:
  - id: module_registration_of_correct_class
    message: $B should be the same as $A
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
              $FUNC(...) {
                ...
                this.$VAR.register($A, new $B(...));
                ...
              }
        - pattern-not-inside: >
              $FUNC(...) {
                ...
                this.$VAR.register($A, new $A(...));
                ...
              }
        - focus-metavariable:
```

```
          - $B

  - id: module_stores_same_index
    message: $INDEX is repeated
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            $FUNC(...) {
              ...
              this.$VAR.register($A, new $A($NAME, $INDEX));
              ...
              this.$VAR.register($B, new $B($NAME, $INDEX));
              ...
            }
        - focus-metavariable:
          - $INDEX
```

Figure G.3: The `module_registration_of_correct_class` and `module_stores_same_index` Semgrep rules

Several rules detect problems in schema definitions:

```
rules:
  - id: schema_min_max_items_without_array
    message: Schema object contains the minItems and/or maxItems properties but is
not of type array
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-either:
            - pattern-inside: >
                {
                    minItems: ...,
                }
            - pattern-inside: >
                {
                    maxItems: ...,
                }
        - pattern-not-inside: >
            {
                type: 'array',
            }
        - pattern-not-inside: >
            {
                ...$X
            }
```

Figure G.4: The `schema_min_max_items_without_array` Semgrep rule

```
rules:
  - id: schema_property_element_without_field_number
```

```
    message: $PROP_A does not have a fieldNumber
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
            }
        - pattern-not-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                        fieldNumber: ...,
                        ...,
                    },
                    ...,
                },
                ...,
            }
        - focus-metavariable:
            - $PROP_A
```

Figure G.5: The `schema_property_element_without_field_number` Semgrep rule

```
rules:
  - id: schema_with_duplicate_field_number
    message: Two schema members have the same field number
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            $SCHEMA = {
                ...,
                $id: $ID,
                ...,
            }
        - pattern-inside: >
            properties: {
              ...,
              $PROP_A: {
                ...,
                fieldNumber: $FIELD_NUMBER,
```

```
                ...,
              },
              ...,
              $PROP_B: {
                ...,
                fieldNumber: $FIELD_NUMBER,
                ...,
              },
              ...,
            }
        - focus-metavariable:
            - $FIELD_NUMBER
```

Figure G.6: The `schema_with_duplicate_field_number` Semgrep rule

```
rules:
  - id: schema_with_field_not_required
    message: Field $PROP_A is not required
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
            }
        - pattern-not-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
                required: [..., '$PROP_A', ...],
                ...,
            }
        - focus-metavariable:
            - $PROP_A
```

Figure G.7: The `schema_with_field_not_required` Semgrep rule

```yaml
rules:
  - id: verify_without_schema_verify
    message: Found a verify function without a call to validator.validate(...)
    languages: [typescript]
    severity: WARNING
    patterns:
        - patterns:
            - pattern-inside: >
                verify(...) {
                  ...
                }
            - pattern-not-inside: >
                verify(...) {
                  ...
                  validator.validate(...);
                  ...
                }
```

Figure G.8: The `verify_without_shema_verify` Semgrep rule

```yaml
rules:
  - id: schema_with_required_that_is_not_a_property
    message: The required property $PROP_A does not exist
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-inside: >
            {
                ...,
                required: [..., '$PROP_A', ...],
                ...,
            }
        - pattern-not-inside: >
            {
                ...,
                properties: {
                    ...,
                    $PROP_A: {
                        ...,
                    },
                    ...,
                },
                ...,
                required: [..., '$PROP_A', ...],
                ...,
            }
        - focus-metavariable:
            - $PROP_A
```

Figure G.9: The `schema_with_required_that_is_not_a_property` Semgrep rule

```yaml
rules:
  - id: schema_hardcoded_pattern
    message: A fastest-validator schema uses a hardcoded pattern instead of a
constant
    languages: [javascript]
    severity: WARNING
    patterns:
      - pattern-inside: >
          {
              params: {
                  $X: { pattern: /.../ },
              }
          }
```

Figure G.10: The `schema_hardcoded_pattern` Semgrep rule

```yaml
rules:
  - id: schema_int_format_with_integer_type
    message: Found a schema property using the ‘integer’ type with a format of
‘int32’, ‘int64’, ‘uint32’, or ‘uint64’. This will not correctly validate integers
due to a bug in lisk-validator.
    languages: [typescript]
    severity: WARNING
    patterns:
        - pattern-either:
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'uint32',
                }
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'uint64',
                }
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'int32',
                }
            - pattern-inside: >
                {
                    type: 'integer',
                    format: 'int64',
                }
```

Figure G.11: The `schema_int_format_with_integer_type` Semgrep rule

Ideas for New Rules

The following are ideas for additional Semgrep rules:

- Detect incorrect calls to the `intToBuffer` function (figure G.12), such as calls with constant values larger than 2^{64}, larger than the provided `byteLength` argument, or with wrong signedness. The function uses `buffer.writeUIntBE` and similar methods, which throw a `RangeError` when invalid arguments are provided. The error may be unexpected during runtime. The rule may benefit from more complex analyses than the ones provided by Semgrep (e.g., from data flow analysis, which is implemented in CodeQL). The rule may detect all calls to methods such as `buffer.writeUIntBE` instead of the `intToBuffer` wrapper.

```
export const intToBuffer = (
	value: number | string,
	byteLength: number,
	endianness = BIG_ENDIAN,
	signed = false,
): Buffer => {
```

Figure G.12: Header of the `intToBuffer` function (`lisk-sdk/elements/lisk-cryptography/src/utils.ts#110–115`)

- Detect all message tags used in the system and ensure that they are unique. More specifically, tags must not have a common prefix pairwise. To detect all tags, calls to the `tagMessage` method may be listed and first arguments to the calls (which should be constants) may be used as the list of tags. While Semgrep can find all tags, post-processing will be needed to test their uniqueness (CodeQL may also be a suitable alternative).

- Check whether a serializable class correctly uses all fields in all relevant places. For example, the `BlockHeader` class defines a set of fields, which must be correctly reflected in its constructor, in its `_getSigningProps` method, in the `BlockHeaderAttrs` interface, and in the `blockHeaderSchema` schema. Because all this code is manually written, developers may easily make mistakes and forget to include a single field in one place.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From August 21 to August 25 and on September 1, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Lisk Foundation team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 100 issues described in this report, the Lisk Foundation team has resolved 90 issues, has partially resolved four issues, and has not resolved the remaining three issues. The fix statuses for three issues are undetermined. For additional information, please see the Detailed Fix Review Results below.

Lisk SDK

ID	Title	Status
1	HTTP and WebSocket API endpoints lack access controls	Resolved
2	Unreachable check for ApplyPenaltyError	Resolved
6	Discrepancies between LIPs and the corresponding implementation	Undetermined
9	Sensitive data stored in world-readable files	Resolved
10	BLS operations involving secret data are not constant time	Resolved
11	LIP-0066 depends on unfinalized EIP-2333 and EIP-2334	Resolved
12	Invalid argon2id memory parameter	Resolved
13	Users cannot set argon2id memory limit	Resolved

14	Log injection via RPC method name	**Resolved**
15	RPC request name allows access to object prototype properties	**Resolved**
16	XSS in the dashboard plugin	**Undetermined**
17	Encryption file format proposed in LIP-0067 weakens encryption	**Resolved**
18	Pre-hashing enables collisions	**Resolved**
19	lisk-codec's decoding method does not validate wire-type data strictly	**Resolved**
20	lisk-codec's decoding method for varints is not strict	**Resolved**
21	lisk-codec's decoding method for strings introduces ambiguity due to UTF-8 encoding	**Resolved**
22	Hash onion computation can exhaust node resources	**Resolved**
23	setHashOnion RPC method does not validate seed length	**Resolved**
24	token methods lack checks for negative token amounts	**Resolved**
25	token module supports all tokens from mainchain and not just LSK	**Resolved**
26	Schemas with required fields of nonexistent properties	**Resolved**
27	Schemas with repeated IDs	**Resolved**
28	Schemas do not require fields that should be required	**Resolved**

29	Lisk Validator allows extra arguments	Unresolved
30	Commands are responsible for validating their parameters	Resolved
31	Insufficient data validation in validators module	Resolved
32	Event indexes are incorrectly converted to bytes for use in sparse Merkle trees	Resolved
33	Lack of bounds on reward configuration could cause node crashes	Resolved
34	Mainchain registration schema validates signature length incorrectly	Resolved
35	Sidechain terminated command uses the wrong chain ID variable	Resolved
36	Hex format validator allows empty and odd-length strings	Resolved
37	CCM fees are always burned	Resolved
38	CCM fees are underspecified and the implementation is not defensive	Resolved
39	Unspecified order for running interoperability modules	Resolved
40	The interoperability module's terminateChain method does not work	Resolved
41	Chain ID length not validated in interoperability endpoints	Resolved
42	Bounced CCM fees are not escrowed	Resolved
43	The send method accepts a status different from OK	Resolved

44	The error method incorrectly checks the status field	Resolved
45	Send method may lead to crash when missing the timestamp parameter	Resolved
46	The channel terminated command does not validate the CCM status	Resolved
47	Invalid use of values in the sparse Merkle tree	Resolved
48	Recovered messages can be replayed	Resolved
49	StateStore handles multiple snapshots dangerously	Resolved
50	Invalid base method used in InitializeStateRecoveryCommand	Resolved
51	Incorrect handling of large integers in regular Merkle tree verification	Resolved
52	Lisk Validator does not validate integer formats when provided as number	Resolved
91	totalWeight may exceed MAX_UINT64 in mainchain registration command verification	Resolved
97	Extraneous feeTokenID option configured for token module	Resolved
99	LIP-0037 specifies ambiguous requirements for tags	Resolved
100	BLS library does not properly check secret key	Resolved

Lisk DB

ID	Title	Status
3	Invalid common prefix method	Resolved
4	Panic due to lack of validation for Proof's bitmap length	Resolved
5	Sparse Merkle tree proof's verification algorithm is invalid	Resolved
7	Lack of length validation for leaf keys in sparse Merkle tree proof verification	Resolved
8	Lack of sparse Merkle tree personalized tree-wide constant	Resolved

Lisk Desktop

ID	Title	Status
53	Electron version is outdated and uses vulnerable Chromium version	Resolved
54	Electron renderer lacks sandboxing	Resolved
55	Lack of a CSP in lisk-desktop	Partially Resolved
56	Electron app does not validate URLs on new windows and navigation	Resolved
57	IPC exposes overly sensitive functionality	Resolved

58	Electron local server is exposed on all interfaces	Resolved
59	Unnecessary use of innerHTML	Resolved
60	ReDoS in isValidRemote function	Resolved
61	Improper handling of the custom lisk:// protocol in lisk-desktop	Resolved
62	Lack of permission checks in the Electron application	Resolved
73	Unnecessary XSS risk in htmlStringToReact	Resolved
80	WalletConnect integration crashes on requiredNamespaces without Lisk	Resolved
81	WalletConnect integration accepts any namespaces	Resolved
82	Impossible to cancel a WalletConnect approval request without refreshing	Resolved
83	Desktop and mobile applications do not validate data coming from online services	Partially Resolved
84	Users can be tricked into unknowingly authorizing a dapp on a chain ID	Resolved
88	Missing round-trip property between parseSearchParams and stringifySearchParams	Resolved
89	Several lisk-desktop identifiers are not unique	Resolved
93	Incorrect entropy in lisk-desktop passphrase generation	Resolved
94	lisk-desktop attempts to open a nonexistent modal	Resolved

95	Phishing risk on the deviceDisconnectDialog modal	Resolved
96	Use of JavaScript instead of TypeScript	Unresolved

Lisk Mobile

ID	Title	Status
68	Mobile iOS application does not use system-managed login input fields	Resolved
69	Mobile iOS application does not exclude keychain items from online backups	Resolved
70	Mobile iOS application does not disable custom iOS keyboards	Resolved
71	Mobile application uses invalid KDF algorithm and parameters	Resolved
72	Mobile iOS application includes redundant permissions	Resolved
74	Mobile iOS application disables ATS on iOS devices	Unresolved
75	Mobile application does not implement certificate pinning	Undetermined
76	Mobile application biometric authentication is prone to bypasses	Resolved
77	Mobile application is susceptible to URI scheme hijacking due to not using Universal Links and App Links	Resolved
78	Mobile iOS application filesystem encryption is not enabled for locked devices	Resolved

79	Mobile Android application permission riding is possible	Partially Resolved
87	Mobile application caches password in transaction-signing form	Resolved
90	Mobile application insufficiently validates and incorrectly displays Amount value in transaction transfer	Partially Resolved
98	Mnemonic recovery passphrase can be copied to clipboard	Resolved

Lisk Service

ID	Title	Status
63	ReDoS in API parameter validation	Resolved
64	HTTP rate-limiting options are not passed to Gateway container	Resolved
65	HTTP rate limiter trusts X-Forwarded-For header from client	Resolved
66	Unhandled exception when filename for transaction history download is a directory	Resolved
67	Path traversal in transaction history download	Resolved
85	Misnamed WebSocket rate-limiting options in Compose file	Resolved
86	Use of hard-coded validation patterns	Resolved
92	Lack of MySQL LIKE escaping in search parameters	Resolved

Detailed Fix Review Results—Lisk SDK

TOB-LISK-1: HTTP and WebSocket API endpoints lack access controls

Resolved in PR #8478. With the fix, the default configuration for a Lisk application (obtained after running `lisk init`) is to not expose any RPC method. To enable RPC methods, developers must update the config and specify the ones they want to enable. To further improve security, the `accessControlAllowOrigin` option should not default to the asterisk symbol [`*`]; instead, it should default to the empty array. This will still allow users to use a tool such as Curl to access the RPC but will limit what websites can access.

TOB-LISK-2: Unreachable check for ApplyPenaltyError

Resolved in PR #8445.

TOB-LISK-6: Discrepancies between LIPs and the corresponding implementation

Undetermined. Due to time constraints and the low severity of these issues, we did not review whether every discrepancy was effectively fixed.

TOB-LISK-9: Sensitive data stored in world-readable files

Resolved in PR #8482.

TOB-LISK-10: BLS operations involving secret data are not constant time

Resolved in PR #8487, PR #8829, PR #8877, PR #8881, and PR #108. The Lisk Foundation team updated vulnerable methods in the code to use constant time comparisons. The team contacted upstream library maintainers to resolve relevant issues in their codebase.

TOB-LISK-11: LIP-0066 depends on unfinalized EIP-2333 and EIP-2334

Resolved. The Lisk team provided the following context:

> *We will track changes to EIPs 2333 and 2334.*

TOB-LISK-12: Invalid argon2id memory parameter

Resolved in PR #8516. The `ARGON2_MEMORY` constant was updated to a larger value.

TOB-LISK-13: Users cannot set argon2id memory limit

Resolved in PR #8498.

TOB-LISK-14: Log injection via RPC method name

Resolved in PR #8513.

TOB-LISK-15: RPC request name allows access to object prototype properties

Resolved in PR #8513.

TOB-LISK-16: XSS in the dashboard plugin

Undetermined. We did not determine whether this issue was fixed because the Lisk team considered the dashboard plugin—a plugin used only during development—to be out of scope. The Lisk team provided the following note:

Dashboard plugin is out of the scope for this audit, and it is not meant for the production usage. We will mention this in the documentation as well.

TOB-LISK-17: Encryption file format proposed in LIP-0067 weakens encryption
Resolved in PR #8746 and PR #400. The Lisk team replaced the 256-bit version of the AES-GCM with the 128-bit version. Therefore, the 16-byte MAC key no longer downgrades the system's security, as the system was explicitly made weaker. While 128 bits of security is considered safe for symmetric encryption, we recommend switching back to 256-bit encryption and enhancing the MAC security or removing the MAC altogether (as integrity is already guaranteed by the AES-GCM tag). This will make the system more future proof.

Moreover, the current MAC is vulnerable to length extension attacks.

The Lisk team provided the following note:

We acknowledge the improvements proposed in the slack, but we prefer to keep it as it is for now and improve in the future

TOB-LISK-18: Pre-hashing enables collisions
Resolved in PR #8805 and PR #429. The double-hash signature scheme was removed. Message signing functionality is now implemented with the single-hash `signDataWithPrivateKey` function and a new, unique tag.

TOB-LISK-19: lisk-codec's decoding method does not validate wire-type data strictly
Resolved in PR #8497.

TOB-LISK-20: lisk-codec's decoding method for varints is not strict
Resolved in PR #8497.

TOB-LISK-21: lisk-codec's decoding method for strings introduces ambiguity due to UTF-8 encoding
Resolved in PR #8497.

TOB-LISK-22: Hash onion computation can exhaust node resources
Resolved in PR #8521.

TOB-LISK-23: setHashOnion RPC method does not validate seed length
Resolved in PR #8521.

TOB-LISK-24: token methods lack checks for negative token amounts
Resolved in PR #8559 and PR #8918. The Lisk team added checks for negative amounts in the vulnerable functions. Ideally, the Lisk team would use a new type that supports only unsigned `BigInts`. We have not researched how this could be accomplished, but JavaScript does not support unsigned `BigInts` by default.

TOB-LISK-25: token module supports all tokens from mainchain and not just LSK
Resolved in PR #8565.

TOB-LISK-26: Schemas with required fields of nonexistent properties
Resolved in PR #8593.

TOB-LISK-27: Schemas with repeated IDs
Resolved in PR #8594.

TOB-LISK-28: Schemas do not require fields that should be required
Resolved in PR #8593 and PR #8917.

TOB-LISK-29: Lisk Validator allows extra arguments
Unresolved. The Lisk team noted the concern and accepted the risk, adding the following remark:

> *We use dynamic nondefined properties on the config, so we cannot enable for all the schemas.*

TOB-LISK-30: Commands are responsible for validating their parameters
Resolved in PR #8558. LIPs were updated in PR #440. With this fix, the Lisk team ensures that the parameters of commands are always verified.

TOB-LISK-31: Insufficient data validation in validators module
Resolved in PR #8920. All methods pointed out in the finding now validate all relevant argument lengths. Length properties were added to the relevant fields of the `ValidateBLSKeyRequest` schema.

TOB-LISK-32: Event indexes are incorrectly converted to bytes for use in sparse Merkle trees
Resolved in PR #8515. The `indexBit` variable was correctly converted to an unsigned integer by using an unsigned right shift by `0: (>>> 0)`.

TOB-LISK-33: Lack of bounds on reward configuration could cause node crashes
Resolved. This finding was a nonissue because there was already an `if` condition that prevented the `config.offset` variable from being greater than the `height` variable.

TOB-LISK-34: Mainchain registration schema validates signature length incorrectly
Resolved in PR #8558.

TOB-LISK-35: Sidechain terminated command uses the wrong chain ID variable
Resolved in PR #8748.

TOB-LISK-36: Hex format validator allows empty and odd-length strings
Resolved in PR #8556.

TOB-LISK-37: CCM fees are always burned
Resolved in PR #8601.

TOB-LISK-38: CCM fees are underspecified and the implementation is not defensive
Resolved in several pull requests. The Lisk team updated and expanded the specification of how fees work in PR #392. In PR #8806, the implementation was made more defensive by creating the `getMessageFeeTokenIDFromCCM` function and by checking, where appropriate, that the token in use is the LSK token. The corresponding LIPs were updated in PR #426 and PR #437.

TOB-LISK-39: Unspecified order for running interoperability modules
Resolved in PR #373.

TOB-LISK-40: The interoperability module's terminateChain method does not work
Resolved in PR #8740.

TOB-LISK-41: Chain ID length not validated in interoperability endpoints
Resolved in PR #8627.

TOB-LISK-42: Bounced CCM fees are not escrowed
Resolved in several pull requests. PR #8765 modifies the code so that bounced CCMs have a fixed fee of `0`; PR #8769 updates how the `apply` function works; PR #8817 updates how the `forward` function works; PR #8839 updates the `token` module's `beforeCCMForwarding` function; and PR #8816 modifies the code responsible for message recovery. The relevant LIPs were updated in PR #405. In summary, the CCM fee logic was updated to give the relayer the whole message fee even in cases where the command is not supported, as explained by the Lisk team:

> *Solving this issue required an extensive update of the CCM processing flow. The main idea now is that the protocol obeys the following rules:*
>
> > *i. The whole amount of the message fee is assigned to the relayer who submitted the CCU transaction that contained the CCM, even in case module/command is not supported (which was not the case before). If the message fee token is native to the receiving chain, the escrow is updated.*
> >
> > *ii. A bounced CCM will always have the message fee set to `0`, since the fee of the original CCM is already assigned to the relayer. This way, no additional escrow updates are required due to bouncing.*
> >
> > *iii. A CCM gets bounced if its message fee is greater than or equal to `minimumFee`. In case a CCM needs to get bounced but the message fee is smaller than `minimumFee`, then it gets discarded.*

This way, even if the CCM gets bounced, the whole message fee is assigned in the original receiving chain and the escrows are updated correctly.

TOB-LISK-43: The send method accepts a status different from OK
Resolved in PR #8690.

TOB-LISK-44: The error method incorrectly checks the status field
Resolved in PR #8689.

TOB-LISK-45: Send method may lead to crash when missing the timestamp parameter
Resolved in PR #8710

TOB-LISK-46: The channel terminated command does not validate the CCM status
Resolved in PR #8713. We still recommend implementing the long-term recommendation to make error handling clearer and remove the burden on developers to remember to verify the `status` field on every cross-chain command. If the Lisk team can forget to add the check, so can outside developers.

TOB-LISK-47: Invalid use of values in the sparse Merkle tree
Resolved in PR #118, PR #8707, PR #387, PR #431, PR #432, and PR #434. The `params.channel` value is now validated to be nonempty. Uses of empty values and other constants in a sparse Merkle tree were documented in relevant LIPs.

TOB-LISK-48: Recovered messages can be replayed
Resolved in PR #8778.

TOB-LISK-49: StateStore handles multiple snapshots dangerously
Resolved in PR #8768.

TOB-LISK-50: Invalid base method used in InitializeStateRecoveryCommand
Resolved in PR #8685.

TOB-LISK-51: Incorrect handling of large integers in regular Merkle tree verification
Resolved in PR #8802.

TOB-LISK-52: Lisk Validator does not validate integer formats when provided as number
Resolved in PR #8743 and PR #8926. The `uint32` and `int32` types are correctly validated when provided as numbers, and `uint64` and `int64` types are correctly validated when provided as strings. Schemas follow this convention, so the issue is fixed. However, we recommend unifying all numeric types to require them to be provided as either strings or numbers. Otherwise, developers may easily make mistakes in the future. Moreover, we recommend updating the relevant Semgrep rule and using it in the CI pipeline for automatic verification.

TOB-LISK-91: totalWeight may exceed MAX_UINT64 in mainchain registration command verification
Resolved in PR #8794.

TOB-LISK-97: Extraneous feeTokenID option configured for token module
Resolved in PR #805. The pull request removes the extraneous config option from the `lisk-core` configurations. We still recommend implementing our long-term recommendation of validating the configs with the already defined config schemas. With these checks in place, other misconfigurations may be detected as well.

TOB-LISK-99: LIP-0037 specifies ambiguous requirements for tags
Resolved in PR #8795 and PR #409.

TOB-LISK-100: BLS library does not properly check secret key
Resolved in PR #8829.

Detailed Fix Review Results—Lisk DB

TOB-LISK-3: Invalid common prefix method
Resolved in PR #111.

TOB-LISK-4: Panic due to lack of validation for Proof's bitmap length
Resolved in PR #112.

TOB-LISK-5: Sparse Merkle tree proof's verification algorithm is invalid
Resolved in PR #432, PR #118, and PR #484. New methods were added to `lisk-db`, making the sparse Merkle tree's API less error-prone. The new APIs and other fixes resolved the reported bugs in the proof verification algorithm.

TOB-LISK-7: Lack of length validation for leaf keys in sparse Merkle tree proof verification
Resolved in PR #109. The code was modified to add a length check to the `proof.queries` keys.

TOB-LISK-8: Lack of sparse Merkle tree personalized tree-wide constant
Resolved in PR #119. The code was modified to include a Lisk-unique constant when hashing leafs and a different one when hashing branches. The Lisk team provided the following context:

> *We modify the constant prefixes to `LSK_SMTL_`\`-ascii-encoded and \``LSK_SMTB_`\`-ascii-encoded. This achieves the same as having global unique constants. This prevents attacks across trees used by systems other than Lisk but does not prevent attacks from other trees in the Lisk ecosystem (e.g., state trees of another Lisk chain). To prevent these types of attacks, the global constant must be specific to the Lisk chain, e.g., one could prepend the chain ID to each message being hashed. However,*

we do not consider it necessary to address this issue since if the assumptions about security of SHA-256 breaks, several parts of the Lisk protocol would require to be updated as well.

Detailed Fix Review Results—Lisk Desktop

TOB-LISK-53: Electron version is outdated and uses vulnerable Chromium version
Resolved in PR #5120. The Lisk team updated the Electron version to v25.2.0. The Lisk team also started using Dependabot to quickly fix dependencies with vulnerabilities. Furthermore, the Lisk team stated that they will update Electron before creating new RC and production releases:

> *We plan to update Electron before tagging RC and production releases.*

We recommend also using Dependabot version updates to ensure that the Electron package is kept up to date (even if the pull requests are merged only before a new release).

TOB-LISK-54: Electron renderer lacks sandboxing
Resolved in PR #5120.

TOB-LISK-55: Lack of a CSP in lisk-desktop
Partially resolved in PR #5151 and PR #5291. The pull requests add a CSP with the `script-src` directive set to `'self' 'unsafe-eval'` and a set of subresource integrity hashes. The `'unsafe-eval'` was kept because it is required by the Ajv dependency. However, the dependency's documentation provides a way to use the dependency without weakening the CSP. This approach should be used. Moreover, the CSP is missing the `'object-src'` directive. We recommend testing the final, deployed CSP with CSP Evaluator.

TOB-LISK-56: Electron app does not validate URLs on new windows and navigation
Resolved in PR #5157 and PR #5287. The issue was fixed by limiting new navigations to the `lisk.com` hostnames.

TOB-LISK-57: IPC exposes overly sensitive functionality
Resolved in PR #5148. This pull request defines a single function for each functionality that needs to be exposed to the renderer process, following the principle of least privilege.

TOB-LISK-58: Electron local server is exposed on all interfaces
Resolved in PR #5149.

TOB-LISK-59: Unnecessary use of innerHTML
Resolved in PR #5152.

TOB-LISK-60: ReDoS in isValidRemote function

Resolved in PR #5265. The issue was resolved by removing the offending function from the codebase, as it was not used.

TOB-LISK-61: Improper handling of the custom lisk:// protocol in lisk-desktop
Resolved in PR #5158 and PR #5158. The code was modified to allow deep links only to the `wallet` hostname and only with specific arguments.

TOB-LISK-62: Lack of permission checks in the Electron application
Resolved in PR #5153.

TOB-LISK-73: Unnecessary XSS risk in htmlStringToReact
Resolved in PR #5161.

TOB-LISK-80: WalletConnect integration crashes on requiredNamespaces without Lisk
Resolved in PR #5155.

TOB-LISK-81: WalletConnect integration accepts any namespaces
Resolved in PR #5155.

TOB-LISK-82: Impossible to cancel a WalletConnect approval request without refreshing
Resolved in PR #5155.

TOB-LISK-83: Desktop and mobile applications do not validate data coming from online services
Partially resolved in PR #5175. The chain ID is displayed for manual verification. However, the chain ID is not syntactically validated and may contain, for example, white spaces that are invisible to users but will impact the signature. The transaction's asset schema is syntactically validated but is not semantically validated—that is, the online service may send a syntactically valid but malicious schema that will impact the signature.

TOB-LISK-84: Users can be tricked into unknowingly authorizing a dapp on a chain ID
Resolved in PR #5168.

TOB-LISK-88: Missing round-trip property between parseSearchParams and stringifySearchParams
Resolved in PR #5171. We ran the fuzzer that we provided in the finding and found no counterexample to the round-trip property.

TOB-LISK-89: Several lisk-desktop identifiers are not unique
Resolved in PR #5170.

TOB-LISK-93: Incorrect entropy in lisk-desktop passphrase generation
Resolved in PR #5172.

TOB-LISK-94: lisk-desktop attempts to open a nonexistent modal
Resolved in PR #5176.

TOB-LISK-95: Phishing risk on the deviceDisconnectDialog modal
Resolved in PR #5174. The modal was removed from the code.

TOB-LISK-96: Use of JavaScript instead of TypeScript
Unresolved. The code still uses JavaScript instead of TypeScript. Newer parts of the code have also not been written in TypeScript.

Detailed Fix Review Results—Lisk Mobile

TOB-LISK-68: Mobile iOS application does not use system-managed login input fields
Resolved in PR #1906. We have not verified whether the autofilled passwords can be accidently saved in the keychain while bypassing `lisk-mobile`'s custom logic for handling passwords.

TOB-LISK-69: Mobile iOS application does not exclude keychain items from online backups
Resolved in PR #1907.

TOB-LISK-70: Mobile iOS application does not disable custom iOS keyboards
Resolved in PR #1912.

TOB-LISK-71: Mobile application uses invalid KDF algorithm and parameters
Resolved in PR #1911 and PR #2037. The `lisk-mobile` application now uses the argon2id algorithm instead of the PBKDF2 algorithm to derive the encryption key from a password.

TOB-LISK-72: Mobile iOS application includes redundant permissions
Resolved in PR #1919.

TOB-LISK-74: Mobile iOS application disables ATS on iOS devices
Unresolved. The Lisk team attempted to fix the issue in PR #1946; however, the `NSAllowsArbitraryLoads` key is still set to `true`, which, as Apple's documentation explains, will "disable App Transport Security (ATS) restrictions for all domains not specified in the `NSExceptionDomains` dictionary." Since the `NSExceptionDomains` list is empty, ATS is disabled for all domains.

TOB-LISK-75: Mobile application does not implement certificate pinning
Undetermined. The target version supplied for the fix review does not include the code from PR #1984, which attempts to add certificate pinning to the application. We have not reviewed the effectiveness of the changes in this pull request.

TOB-LISK-76: Mobile application biometric authentication is prone to bypasses
Resolved in PR #1918.

TOB-LISK-77: Mobile application is susceptible to URI scheme hijacking due to not using Universal Links and App Links
Resolved in PR #1941. Lisk on iOS now supports Universal Links and Lisk on Android now supports App Links. The old `lisk://` URL schemes are still supported in parallel, but the Lisk team indicated that these will be removed in the future.

TOB-LISK-78: Mobile iOS application filesystem encryption is not enabled for locked devices
Resolved in PR #1913.

TOB-LISK-79: Mobile Android application permission riding is possible
Partially resolved in PR #1909. The data access auditing feature is used to log events of users' private data being accessed by the Lisk application. However, the feature is not used to prevent third-party dependencies from misusing permissions granted to `lisk-mobile`.

TOB-LISK-87: Mobile application caches password in transaction-signing form
Resolved in PR #1925. The password is now correctly cleared from memory. A transaction is validated before the user is asked for the password.

TOB-LISK-90: Mobile application insufficiently validates and incorrectly displays amount value in transaction transfer
Partially resolved. The `validateTransactionAmount` function was introduced in PR #1929 to make sure the `amount` parameter is properly validated before displaying it to the user. However, this function is used only in the `useSendTokenAmountChecker` function to decide if `0` or the user-provided token amount should be checked against the user balance. The user is still shown the user-provided value in the UI without validation or sanitization.

TOB-LISK-98: Mnemonic recovery passphrase can be copied to clipboard
Resolved in PR #1954.

Detailed Fix Review Results—Lisk Service

TOB-LISK-63: ReDoS in API parameter validation
Resolved in PR #1721.

TOB-LISK-64: HTTP rate-limiting options are not passed to Gateway container
Resolved in PR #1719.

TOB-LISK-65: HTTP rate limiter trusts X-Forwarded-For header from client
Resolved in PR #1728.

TOB-LISK-66: Unhandled exception when filename for transaction history download is a directory
Resolved in PR #1726.

TOB-LISK-67: Path traversal in transaction history download
Resolved in PR #1726.

TOB-LISK-85: Misnamed WebSocket rate-limiting options in Compose file
Resolved in PR #1719.

TOB-LISK-86: Use of hard-coded validation patterns
Resolved in PR #1721.

TOB-LISK-92: Lack of MySQL LIKE escaping in search parameters
Resolved in PR #1734. The Lisk team fixed this issue by still allowing users to use wildcards if they pass the `allowWildCards` parameter in the GET request. To further increase code quality, we recommend creating a function that sanitizes the pattern, instead of copying the `.replace(/%/g, '\\%').replace(/_/g, '\\_')}` code snippet in several locations.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.