Lisk

Security Assessment (Summary Report)

April 9, 2024

Prepared for:
Jan Hackfeld
Lisk

Prepared by: **Elvis Skoždopolj, Justin Jacob, and Vasco Franco**

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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

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

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Vasco Franco, Consultant
vasco.franco@trailofbits.com

Project Timeline

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

Date	Event
February 23, 2024	Pre-project kickoff call
March 11, 2024	Status update meeting
March 19, 2024	Delivery of report draft
March 19, 2024	Report readout meeting
April 9, 2024	Delivery of summary report

Project Targets

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

lisk-contracts

Repository	https://github.com/LiskHQ/lisk-contracts
Version	`e39b6651855229c652a7f4965740cc601334d871`
Type	Solidity
Platform	Ethereum

lisk-token-claim

Repository	https://github.com/LiskHQ/lisk-token-claim
Version	`3d49c8886470309b944f98400e8931098059b299`
Type	TypeScript
Platform	Node.js

Executive Summary

Engagement Overview

Lisk engaged Trail of Bits to review the security of the smart contracts and deployment scripts in the `lisk-contracts` repository at commit `e39b665` and the off-chain components in the `lisk-token-claim` repository at commit `3d49c88`. The smart contracts consist of two ERC-20 token contracts and a `L2Claim` contract, which allows users of the original Lisk blockchain to claim their token allocations on Lisk's new Optimism-based L2 blockchain, by providing an EdDSA signature and a Merkle proof to verify the legitimacy of their claims.

The off-chain codebase consists of two components: the `tree-builder`, which uses a locally stored database with the state of the Lisk blockchain and builds a Merkle tree whose root is then passed to the L2 smart contract, and the `claim-backend`, where users can check their eligibility for a token claim and store and receive signatures for multisignature accounts.

A team of three consultants conducted the review from March 6 to March 15, 2024, for a total of three engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the targets, using automated and manual processes.

Observations and Impact

We reviewed the smart contracts for common Solidity pitfalls, such as issues related to Merkle proof verification, signature verification, reentrancy attacks, insufficient or incorrect validation, front-running resistance, and missing or insufficient access controls. The Ed25519 Solidity implementation was considered out of scope for this review.

We reviewed the off-chain components to check that all accounts are included in the snapshot; that each account's information is encoded correctly (e.g., amounts of tokens are not truncated, addresses are properly encoded); that the Merkle tree library is used correctly; that the `claim-backend` does not allow a single user to fill up the database, which could lead to resource exhaustion; and that the `claim-backend` endpoints are overall not vulnerable to external attackers. We assumed the `lisk-db` `iterate` function works as intended (i.e., that it returns each element that is between the `gte` and `lte` values).

In the smart contracts, we identified one high-severity issue related to the `claimMultisigAccount` function: it is missing a zero-value check on the length of the signature array, which allows users to skip the signature verification and drain all regular account token allocations. Additionally, we identified one low-severity issue and three informational-severity issues related to a missing event emission, insufficient validation, and the lack of a two-step process for a critical operation.

We identified one informational-severity issue related to a trivial loop condition in the off-chain components.

Recommendations

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

- **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.
- **Improve the testing suite.** Tests should cover both happy and unhappy paths as well as edge cases throughout the system. Consider using advanced testing techniques such as smart contract fuzzing to further test various parts of the codebase.
- **Improve the user-facing documentation.** Comprehensive user-facing documentation detailing the claiming process, the powers and privileges of the administrator, and how and when privileged operations such as contract upgrades will be initiated would help build trust in the system.
- **Consider requiring that token claims be made from the signed recipient address in the smart contract claim functions.** This will ensure that users will not mistakenly have claimed tokens sent to addresses that they do not control. The Lisk team currently plans to enforce this condition using the front-end UI.
- **Consider adding additional checks around code that reads and decodes values from files in the deployment scripts.** Data being written to and read from files could be misinterpreted, leading to incorrect deployment or loss of assets when transferring tokens. Ensure that data encoding and decoding is thoroughly tested with the full range of expected types and values.

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.

Category	Summary	Result
Arithmetic	The codebase uses a modern version of the Solidity compiler with built-in checked arithmetic. No complex or unchecked arithmetic operations are present in the in-scope contracts. There is one instance of `uint` downcasting of a user-controlled parameter in the `claimMultisigAccount` function; however, if the value overflows, the function will revert when verifying the Merkle proof, making this operation safe.	**Strong**
Auditing	Most state-changing functions emit events; however, we identified one function that sets an important system parameter that does not emit an event. The system could benefit from the addition of an event emission to this function. Also, we are not aware of any off-chain monitoring system or incident response plan in place.	**Moderate**
Authentication / Access Controls	The `L1LiskToken` contract uses role-based access controls to divide privileges between the administrator role and the burner role. The `L2Claim` contract defines one role that is able to modify system parameters and upgrade the contract (the owner). Both contracts use a single-step process for the transfer of ownership roles, which can be error-prone. The system could benefit from use of two-step processes for all irrecoverable critical operations, as well as additional user-facing documentation that details the nature of the roles and when certain privileged actions will be taken.	**Moderate**
Complexity Management	The system has a simple architecture with well-defined components and minimal code duplication. A clear and consistent naming convention is used throughout the contracts.	**Satisfactory**

Decentralization	The `L2Claim` contract, which holds and distributes the L2 LSK tokens, is upgradeable by a single entity, and users are not able to opt out of upgrades. Furthermore, there is no documentation about upgrades or the powers and privileges of the administrator.	**Weak**
Documentation	The contracts contain inline and NatSpec documentation, which makes it easy to understand developer intentions. The system setup and deployment is clearly documented in the repository documentation. However, the user-facing documentation could have more detail regarding the claiming process and the privileged roles and operations of the system.	**Satisfactory**
Low-Level Manipulation	The in-scope contracts do not use any low-level manipulation.	**Strong**
Testing and Verification	Unit tests are thorough in testing the expected system functionality; however, the testing suite would benefit from the addition of common adversarial situations such as the passing of unvalidated data. A deeper testing suite would have uncovered issues such as TOB-LSK-1 and TOB-LSK-5. Furthermore, mutation testing uncovered some minor gaps in the testing of view functions and event emissions. There should be additional testing done around these areas to ensure intended behavior throughout the system.	**Moderate**
Transaction Ordering	No transaction ordering risks were identified during the review.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Regular user token allocations can be stolen	Data Validation	**High**
2	Lack of two-step process for ownership transfer	Data Validation	**Low**
3	setDAOAddress does not emit an event	Auditing and Logging	**Informational**
4	Trivial loop condition in snapshot creation	Data Validation	**Informational**
5	Lack of zero-value checks in the initialize function	Data Validation	**Informational**

A. 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
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

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.

B. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Lisk will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

C. Code Quality Findings

The following findings are not associated with specific vulnerabilities. However, addressing them may enhance code readability and prevent the introduction of vulnerabilities in the future.

- **Use of incorrect naming convention for some non-constant variables.** The `BRIDGE` state variable of the `L2LiskToken` contract uses the naming convention for constants; however, this variable is mutable. Use camelCase for all mutable state variables.
- **Missing proof length validation.** While this does not lead to a security vulnerability in the current codebase, it is good practice to explicitly validate all edge cases so that the codebase security is robust against any future code changes that might make this a viable attack vector.
- **Use of unsafe casting.** There is one instance of unsafe casting in the `claimMultisigAccount` function of the `L2Claim` contract. Even though this is not an issue since the function will revert on overflow, using a library such as OpenZeppelin's `SafeCast` to perform checked casting operations will prevent an overflow from occurring in the first place.

D. Mutation Testing

This appendix outlines how we conducted mutation testing and highlights some of the most actionable results.

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in the test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We used an experimental new mutation tool, `slither-mutate`, to conduct our mutation testing campaign. This tool is custom-made for Solidity and features higher performance and fewer false positives than existing tools such as `universalmutator`.

Although this tool has not been formally released yet, it is open source and available for review in the `slither/tools/mutator` subdirectory on the `dev` branch of the `slither` repository.

The mutation campaign was run against the smart contracts using the following commands:

```
slither-mutate src --test-cmd="forge test" --ignore-dirs='data,utils'
```

Figure D.1: A bash script that runs a mutation testing campaign against each Solidity file in the `src` directory

Consider the following notes about the above commands:

- The overall runtime is approximately 1 hour on a consumer-grade laptop.
- The `--test-cmd` flag specifies the command to run to assess mutant validity. A `--fail-fast` flag will automatically be added to forge test commands to improve runtime.
- The `--ignore-dirs` flag specifies which directories will be ignored.

An abbreviated, illustrative example of a mutation test output file is shown in figure D.2.

```
INFO:Slither-Mutate:Mutating contract L2Claim
INFO:Slither-Mutate:[LIR] Line 190: 'uint256 i = 0' ==> 'uint256 i = 0' --> UNCAUGHT
INFO:Slither-Mutate:[LIR] Line 203: 'uint256 i = 0' ==> 'uint256 i = 0' --> UNCAUGHT
INFO:Slither-Mutate:[LIR] Line 213: 'uint256 i = 0' ==> 'uint256 i = 0' --> UNCAUGHT
INFO:Slither-Mutate:[MVIV] Line 190: 'uint256 i = ' ==> 'uint256 i ' --> UNCAUGHT
INFO:Slither-Mutate:[MVIV] Line 203: 'uint256 i = ' ==> 'uint256 i ' --> UNCAUGHT
```

Figure D.2: Abbreviated output from the mutation testing campaign on `L2Claim.sol`

In summary, the following is a subset of features of the contracts that lack sufficient tests:

- The `ClaimingEnded`, `LSKClaimed`, `Mint`, and `Burn` events are not tested.
- The `L2Claim` contract's `claimMultisigAccount` function is not tested with various signature lengths.
- The `recoverLSK` function is not tested with various timestamps.
- The `supportsInterface` function is not tested.

We recommend that the Lisk team review the existing tests and add additional verification that will catch the aforementioned types of mutations.

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.

On March 28, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Lisk team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Lisk has resolved all five issues disclosed in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Regular user token allocations can be stolen	**Resolved**
2	Lack of two-step process for ownership transfer	**Resolved**
3	setDAOAddress does not emit an event	**Resolved**
4	Trivial loop condition in snapshot creation	**Resolved**
5	Lack of zero-value checks in the initialize function	**Resolved**

Detailed Fix Review Results

TOB-LSK-1: Regular user token allocations can be stolen
Resolved in PR #58. Validation was added to the `claimMultisigAccount` function to require that the `_sigs` array, containing the Ed25519 signatures, cannot be of zero length. This ensures that at least one multisignature key needs to be provided in the Merkle leaf data, preventing collisions with regular account leaves.

TOB-LSK-2: Lack of two-step process for ownership transfer
Resolved in PR #59. A two-step process for ownership transfer was added to the `L1LiskToken` and `L2Claim` contracts. The `L1LiskToken` and `L2Claim` deployment scripts and unit tests were updated to reflect and test this change.

TOB-LSK-3: setDAOAddress does not emit an event
Resolved in PR #60. An event emission was added to the `setDAOAddress` function of the `L2Claim` contract. Additional event emission verification was added to the `L2Claim` unit tests.

TOB-LSK-4: Trivial loop condition in snapshot creation
Resolved in PR #17. The trivial loop condition in the snapshot creation process was removed.

TOB-LSK-5: Lack of zero-value checks in the initialize function
Resolved in PR #66. Zero-value checks were added to all input parameters of the `initialize` function of the `L2Claim` contract. Additional unit tests were added to test this validation.

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.