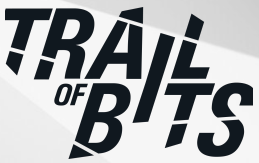

Lisk SDK 6.1 Sapphire, NFT and PoA modules

Security Assessment

November 27, 2023

Prepared for:
Oliver Beddows
Lisk Foundation

Prepared by: **Vasco Franco, Sam Alws, and Paweł Płatek**

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

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 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:

Vasco Franco, Consultant
vasco.franco@trailofbits.com

Sam Alws, Consultant
sam.alws@trailofbits.com

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Project Timeline

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

Date	Event
August 25, 2023	Pre-project kickoff call
September 1, 2023	Status update meeting #1
September 11, 2023	Delivery of comprehensive report draft
September 11, 2023	Report readout meeting
November 27, 2023	Delivery of comprehensive report

Executive Summary

Engagement Overview

Lisk Foundation engaged Trail of Bits to review the security of the Lisk SDK v6.1 NFT and PoA modules. The NFT module implements the Lisk protocol's logic for handling NFTs, including intra- and cross-chain transfers. The PoA module manages and updates the validator list for proof-of-authority chains.

A team of three consultants conducted the review from August 28 to September 11, 2023, for a total of five engineer-weeks of effort. Our testing efforts focused on the NFT and PoA modules of the Lisk SDK v6.1. With full access to source code and documentation, we performed static and dynamic testing of these Lisk SDK modules, using automated and manual processes.

Observations and Impact

During the audit, we found two issues of medium severity:

- TOB-LISK2-14: Users may bypass an NFTs creation fee by faking bounced messages.
- TOB-LISK2-10: The function that removes all supported NFTs may fail to remove support for all NFTs.

We also found a number of low-severity and informational findings, most related to data validation.

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.

- **Improve testing.** While unit testing coverage is good, we recommend adding fuzz testing to some functionality, as is recommended in finding TOB-LISK2-15, and including the static analysis rules we provided in the first phase of the audit, which would help find issues such as TOB-LISK2-8. Finally, edge cases (e.g., minimum and maximum integer values in schema encoding and decoding) should be more thoroughly tested.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	2
Low	6
Informational	9
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	2
Configuration	3
Data Validation	11
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the Lisk SDK NFT and PoA modules. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the NFT module implementation match its LIP specification?
- Can attackers arbitrarily mint or destroy NFTs?
- Can attackers transfer NFTs intra- or cross-chain without owning the NFT?
- Can attackers bypass paying the NFT creation fee?
- Does the PoA module implementation match its LIP specification?
- Can attackers take control of a PoA chain?
- Can PoA signatures be modified or reused to perform unexpected authority updates?
- Are PoA validator list updates performed correctly and safely?

Project Targets

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

Lisk SDK

Repository	https://github.com/LiskHQ/lisk-sdk/releases/tag/v6.1.0-beta.0
Version	`ed5649eb954c7c47e11eb2d2ea2b84b9336c4c4b`
Type	TypeScript
Platform	Any

Lisk Improvement Proposals (LIPs)

Repository	https://github.com/LiskHQ/lips
Version	`c7d59b177cdd74553d5995298afce3a835dd67f7`
Type	Documentation
Platform	N/A

Project Coverage

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

- Manual review of the NFT and PoA modules, comparing the implementation against the LIP specifications
- Use of static analysis tools such as Semgrep and CodeQL on the NFT and PoA modules (including the Semgrep rules provided in the last audit of the Lisk SDK)
- Dynamic review of some of the NFT and PoA module's functionality with the use of the provided Lisk SDK examples
- A brief manual review of the codepaths in `lisk-codec`, `lisk-cryptography`, and `lisk-validator` relevant to PoA and NFT functionality

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 D
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D

Test Results

The results of this focused testing are detailed below.

Property	Tool	Result
An event class does not have a function caller `error` that adds nonpersistent events (`error_event_is_not_revert` rule).	Semgrep	TOB-LISK2-9
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	**Passed**
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	**Passed**
A schema does not require a property that does not exist (`schema_with_required_that_is_not_a_property` rule).	Semgrep	**Passed**
Schemas use the format attribute only on non-integer types (`schema_int_format_with_integer_type` rule).	Semgrep	**Passed**

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	When necessary, bounds are checked by comparing numbers to `INT_MAX` and `INT_MIN` values. 64-bit integer arithmetic avoids overflows by using the `BigInt` type. We found that the schema validator uses an incorrect value for the `MIN_SINT_32` constant in TOB-LISK2-7.	**Satisfactory**
Auditing	Log quality would be improved if the PoA `RegisterAuthority` and `UpdateGeneratorKey` commands were to emit events during execution. We also found that in the NFT module, some error events were not persisted (TOB-LISK2-8), and that bounced NFT transfer events contained incorrect and insufficient data (TOB-LISK2-12).	**Weak**
Authentication / Access Controls	Authentication for commands is typically done via checks on the transaction's sender; we found no issues related to this.	**Satisfactory**
Complexity Management	Overall, PoA and NFT code is well-organized according to a standard structure. In the NFT module, we found that there was duplicated code of critical functionality (TOB-LISK2-2 and TOB-LISK2-13), and extensive use of magic values (TOB-LISK2-4), which made the code more complex than necessary.	**Moderate**
Cryptography and Key Management	No cryptography is done in the NFT module. When cryptography is done in the PoA module, libraries from other parts of the `lisk-sdk` repository are used, which are out-of-scope for this audit.	**Satisfactory**

Data Handling	We found multiple issues related to data validation: where it is too strict (TOB-LISK2-1), where it should be stricter (TOB-LISK2-3, TOB-LISK2-6, TOB-LISK2-10, TOB-LISK2-13, TOB-LISK2-14, TOB-LISK2-15, TOB-LISK2-16, TOB-LISK2-17), and where it could be done in a more robust way (TOB-LISK2-5). Most of these are low-severity or informational issues, but some have a larger practical outcome: users will sometimes be unable to send NFTs cross-chain (TOB-LISK2-1), users may bypass NFT fee payments (TOB-LISK2-14), and the `removeSupportAllNFTs` may not remove support for all NFTs (TOB-LISK2-10). We did not find any issues where insufficient data validation allows for a loss of NFTs or a loss of control over a PoA chain.	**Moderate**
Documentation	Both modules have LIPs that describe their operation in detail. These LIPs are generally well-written, and the module code generally matches up with the corresponding pseudocode in the LIPs; however, there are some small exceptions (see appendix C).	**Satisfactory**
Maintenance	The only direct dependencies of the NFT and PoA modules are on other parts of the `lisk-sdk` codebase, which was covered in our previous audit with Lisk and was therefore not covered in this audit. The PoA module indirectly (through `@liskhq/lisk-cryptography`) uses the `@chainsafe/blst` NodeJS library for signature verification; we found that the current version (0.2.9) of this library is used, but did not verify the security of this library.	**Satisfactory**
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. Errors are appropriately handled, typically by throwing exceptions or by exiting early and logging an error event. We found a single issue where a malformed NFT that is being recovered may cause an unexpected crash (TOB-LISK2-15).	**Satisfactory**
Testing and	The NFT and PoA modules are thoroughly tested. Test	**Satisfactory**

Verification	coverage is collected on each commit using Codecov; currently, on the development branch, codecov shows the modules having 97% and 94% coverage, respectively. The use of TypeScript verifies that JS dynamic type errors are avoided. However, our brief examination of the tests revealed a bug that made one of the tests ineffective (TOB-LISK2-11). We also found that edge cases were not always thoroughly tested (TOB-LISK2-1, TOB-LISK2-10, TOB-LISK2-17, and more). Furthermore, we think the Semgrep rules and fuzzers that we provided during the phase one audit should be included in the project's CI/CD pipeline. Using these rules, we found one bug (TOB-LISK2-8) that could have been prevented. Finally, we think fuzzing should be implemented for some functionality where it would help to find edge cases (e.g., TOB-LISK2-15 and TOB-LISK2-17).	

Summary of Findings

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

ID	Title	Type	Severity
1	Impossible to send cross-chain NFT transfers in certain configurations	Data Validation	Low
2	NFT module has repeated transfer functionality	Configuration	Low
3	NFT lock method does not check if module is NFT_NOT_LOCKED	Data Validation	Informational
4	NFT methods API could be improved	Configuration	Informational
5	Checks done manually instead of using schema validation	Data Validation	Informational
6	Update authority command allows validators to have 0 weight	Data Validation	Informational
7	Incorrect MIN_SINT_32 constant	Data Validation	Informational
8	Event errors are reverted	Auditing and Logging	Low
9	Unspecified and poorly named NFT endpoints	Configuration	Informational
10	The removeSupportAllNFTs function may not remove support for all NFTs	Data Validation	Medium
11	Bug in removeSupportAllNFTs tests	Testing	Informational
12	Bounced NFT transfers may cause events to contain incorrect data	Auditing and Logging	Low

13	An NFT's attributesArray can have duplicate modules	Data Validation	**Low**
14	Bounced messages may be abused to bypass NFT fees	Data Validation	**Medium**
15	NFT recover function may crash	Data Validation	**Informational**
16	NFT recover function does properly validate NFT attributes array	Data Validation	**Informational**
17	The codec.encode function encodes larger than expected integers	Data Validation	**Low**

Detailed Findings

1. Impossible to send cross-chain NFT transfers in certain configurations	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK2-1
Target: `lisk-sdk/framework/src/modules/nft/method.ts`	

Description

The `isNFTSupported` function—a function called in the CrossChainTransferCommand's execute function when receiving NFTs from foreign chains to check if an NFT is supported—may return incorrect results because the `getCollectionID` function (figure 1.1) enforces that an NFT must be stored before returning the corresponding collection ID.

```
public async getCollectionID(
	methodContext: ImmutableMethodContext,
	nftID: Buffer,
): Promise<Buffer> {
	const nftStore = this.stores.get(NFTStore);
	const nftExists = await nftStore.has(methodContext, nftID);
	if (!nftExists) {
		throw new Error('NFT substore entry does not exist');
	}
	return nftID.slice(LENGTH_CHAIN_ID, LENGTH_CHAIN_ID + LENGTH_COLLECTION_ID);
}
```

Figure 1.1: The `getCollectionID` function (`lisk-sdk/framework/src/modules/nft/method.ts#187–197`)

The `isNFTSupported` function calls `getCollectionID` if: a) the NFT ID is not native to the chain, b) the `SupportedNFTsStore` store does not have the `ALL_SUPPORTED_NFTS_KEY` key, and c) the `supportedCollectionIDArray` array for the given chain is not empty (indicating that all collections are supported). Under these conditions, `isNFTSupported` will always throw an exception and prevent the cross chain transfer.

Exploit Scenario

A developer configures chain A to support NFTs of collection 1 from chain B. A user from chain B attempts to transfer to chain A an NFT belonging to collection 1 of chain B. The cross-chain message (CCM) fails and the NFT cannot be transferred.

Recommendations

Short term, remove the NFT existence check from the `getCollectionID` function. This will allow obtaining the collection ID of non-stored NFT IDs, which is necessary in the `isNFTSupported` function.

Long term, create tests that cover this case and all other possible configurations of the `SupportedNFTsStore` store.

2. NFT module has repeated transfer functionality	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK2-2
Target: NFT module	

Description
The NFT module's transfer command functions, `verify` and `execute`, together have the same checks and function calls as the `transfer` method. Their difference is in how they handle errors. Having duplicated code, especially for critical functionality such as transferring NFTs, is bad practice, as the code may diverge and lead to exploitable bugs.

The same problem exists on the `TransferCrossChainCommand command` and `transferCrossChain` method.

Exploit Scenario
A developer fixes a critical vulnerability in one of the functions with copied functionality, but forgets to fix the counterpart. The code remains vulnerable. An attacker finds out and steals NFTs from users.

Recommendations
Short term, factorize the code into a single function that verifies if a transfer is valid. The common function should throw custom exceptions based on the type of error that occurred. In both the transfer command and method, have the code call the single function and handle errors appropriately (e.g., by just letting the exception go through, or by catching it and emitting an event).

Long term, review other modules for duplicated functionality, especially in critical code paths such as the code that handles transfers. If the code is duplicated, factorize into a single function.

3. NFT lock method does not check if module is NFT_NOT_LOCKED	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK2-3
Target: `lisk-sdk/framework/src/modules/nft/method.ts#L339-L397`	

Description
The NFT lock method does not check that the module attempting to lock the NFT is not equal to the magic value that represents an unlocked NFT, the `NFT_NOT_LOCKED` constant (which is equal to `'nft'`). It is unlikely that another module is or can be named `'nft'`; however, this check should be added to prevent misuse of the `lock` method.

Exploit Scenario
Another module that extends or makes use of the NFT module calls the `lock` method with the module argument set to `'nft'` by mistake (e.g., a developer misunderstands the purpose of the module argument). The NFT does not get locked as the developer expected.

Recommendations
Short term, add a check in the code that ensures that the `lock` function's `module` argument is not equal to `NFT_NOT_LOCKED`.

Long term, review stores that use a single state variable to represent more than one state through the use of "magic" values. In the example described above, the `lockingModule` state variable is used to determine which module locked the NFT and has a special value—`NFT_NOT_LOCKED`—to indicate that no module locked it. If possible, we recommend storing the state in two variables, `isLocked` and `lockingModule`, to reduce the use of "magic values" that may lead to state confusion. Do the same decomposition for other stores, where applicable. If these changes are not possible or wanted, ensure that the magic values are always checked when the state is modified.

4. NFT methods API could be improved

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK2-4
Target: NFT methods	

Description

The NFT module performs various checks, such as if an NFT is escrowed or if an NFT is locked in a way that makes the code longer and less readable. Some checks (e.g., NFT existence) also happen more than necessary because of the way the API is designed.

To check if an NFT is escrowed to another chain, the code does the check shown in figure 4.1 in several locations.

```
if (owner.length === LENGTH_CHAIN_ID) {
```

Figure 4.1: An `if` statement that checks if an NFT is escrowed (`lisk-sdk/framework/src/modules/nft/commands/transfer.ts#60`)

This check relies on the fact that the `owner` field will have a different length if it is storing the address of a native user or the chain ID of a foreign chain. To someone not familiar with the inner workings of the NFT module (e.g., a new employee or a curious user), this is not clear. Instead, the code should call a function named `isNFTEscrowed` to clarify the purpose of the check being performed.

To check if an NFT is locked, the code does the check shown in figure 4.1 in several locations.

```
const lockingModule = await this._method.getLockingModule(
       context.getMethodContext(),
       params.nftID,
);
if (lockingModule !== NFT_NOT_LOCKED) {
```

Figure 4.2: An `if` statement that checks if an NFT is locked (`lisk-sdk/framework/src/modules/nft/commands/transfer.ts#68-75`)

This check relies on the fact that the `lockingModule` has a “magic value,” `NFT_NOT_LOCKED`, that represents an unlocked NFT. Again, to someone not familiar with the inner workings of the NFT module, it may not be clear why the code is checking the

`lockingModule` to see if an NFT is locked. Instead, the code could call a function named `isNFTLocked`. This would also shorten the code's length.

Another problem with the current API is that all NFT methods take an NFT ID argument; as a result, many functions end up performing repeated operations when called in succession. One example of this is the `verify` function of the `TransferCommand` command (figure 4.3).

```
public async verify(context: CommandVerifyContext<Params>):
Promise<VerificationResult> {
	const { params } = context;

	const nftStore = this.stores.get(NFTStore);
	const nftExists = await nftStore.has(context, params.nftID);
	if (!nftExists) {
		throw new Error('NFT substore entry does not exist');
	}

	const owner = await this._method.getNFTOwner(context.getMethodContext(),
params.nftID);
	if (owner.length === LENGTH_CHAIN_ID) {
		throw new Error('NFT is escrowed to another chain');
	}

	if (!owner.equals(context.transaction.senderAddress)) {
		throw new Error('Transfer not initiated by the NFT owner');
	}

	const lockingModule = await this._method.getLockingModule(
		context.getMethodContext(),
		params.nftID,
	);
	if (lockingModule !== NFT_NOT_LOCKED) {
		throw new Error('Locked NFTs cannot be transferred');
	}

	return { status: VerifyStatus.OK, };
}
```

Figure 4.3: The verify function of the `TransferCommand` command (shortened for brevity) (`lisk-sdk/framework/src/modules/nft/commands/transfer.ts#45–80`)

The code from figure 4.3 calls the `nftStore.has` function to check if the NFT exists. Then, the `getNFTOwner` call internally calls `nftStore.has` again to check if the NFT exists and it calls `nftStore.get` to obtain the data of the NFT and extract its owner. Finally, `getLockingModule` will internally call `getNFTOwner` (which calls `nftStore.has` and `nftStore.get` again). In conclusion, in these three operations—checking the existence of the NFT, getting its owner, and getting its locking mode—the existence check is performed three separate times, and the NFT data is obtained two times.

If the API was designed to receive the NFT data object instead of the NFT ID for most operations (all except checking an NFT's existence and getting the NFT), then this duplication of checks would not occur.

In figure 4.4, we suggest an alternative implementation that calls the `isNFTEscrowed` and `isNFTLocked` for shorter and cleaner code. This implementation passes to these functions the NFT object instead of the NFT ID to avoid the duplication of checks described above.

```
public async verify(context: CommandVerifyContext<Params>):
Promise<VerificationResult> {
	const { params } = context;

	const ctx = context.getMethodContext();
	const nftData = await this._method.getNFT(ctx, params.nftID);
	if (!nftData) {
		throw new Error('NFT substore entry does not exist');
	}

	if (this._method.isNFTEscrowed(ctx, nftData)) {
		throw new Error('NFT is escrowed to another chain');
	}

	if (!nftData.owner.equals(context.transaction.senderAddress)) {
		throw new Error('Transfer not initiated by the NFT owner');
	}

	if (this._method.isNFTLocked(ctx, nftData)) {
		throw new Error('Locked NFTs cannot be transferred');
	}

	return { status: VerifyStatus.OK, };
}
```

Figure 4.4: An alternative example implementation of the verify function of the `TransferCommand` command

Recommendations

Short term, create methods to check if an NFT is escrowed and if an NFT is locked. This will improve the code's readability and reduce its size. Furthermore, consider updating most methods of the NFT module to receive the NFT object instead of the NFT ID. This will reduce the amount of time the same check is done in each function.

Long term, review other modules where these recommendations may also apply.

5. Checks done manually instead of using schema validation

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK2-5
Target: `lisk-sdk/framework/src/modules/poa/schemas.ts`	

Description

In two locations in the proof of authority module, checks are done manually when schema validation could be used instead. Using schema validation would result in more robust code.

In `endpoint.ts`, the parameter given to `getValidator` is checked in the following way:

```
const { address } = context.params;
if (typeof address !== 'string') {
	throw new Error('Parameter address must be a string.');
}
cryptoAddress.validateLisk32Address(address);
```

Figure 5.1: Check in `getValidator` (`lisk-sdk/framework/src/modules/poa/endpoint.ts#33-37`)

However, the following check could be performed instead:

```
validator.validate(getValidatorRequestSchema, address)
```

Figure 5.2: Proposed check for `getValidator`

In `commands/update_authority.ts`, the length of the command's `newValidators` list is checked in the following way:

```
if (newValidators.length < 1 || newValidators.length > MAX_NUM_VALIDATORS) {
	throw new Error(
		`newValidators length must be between 1 and ${MAX_NUM_VALIDATORS}
(inclusive).`,
	);
}
```

Figure 5.3: Length check in `updateAuthority` (`lisk-sdk/framework/src/modules/poa/commands/update_authority.ts#33-37`)

However, if `minItems` and `maxItems` fields were added to the schema for this command (`updateAuthoritySchema`), this check would be done automatically by the schema validator.

Recommendations

Short term, add the suggested `validator.validate` check to `endpoint.ts` and add the `minItems` and `maxItems` fields to the `updateAuthoritySchema`.

Long term, ensure that all Lisk schemas are as expressive as possible.

6. Update authority command allows validators to have 0 weight

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK2-6
Target: `lisk-sdk/framework/src/modules/poa/commands/update_authority.ts`	

Description

When the proof of authority module initializes its genesis state, it checks that all active validators have nonzero (and positive) weights:

```
// Check that the weight property of every entry in the
snapshotSubstore.activeValidators array is a positive integer.
if (activeValidators[i].weight <= BigInt(0)) {
       throw new Error(``activeValidators` weight must be positive integer.');
}
```

Figure 6.1: Validator weight check in genesis initialization (`lisk-sdk/framework/src/modules/poa/module.ts#247-250`)

However, the same check is not made by the update authority command.

This means that the current validators could vote to approve a new validator list that includes validators with 0 weight. We could not find any way that this could be exploited to cause a security problem. However, it would be a good practice to ensure that the invariant "`for each i, activeValidators[i].weight > 0`" always holds.

Recommendations

Short term, add a check which throws an exception if a validator has 0 weight.

Long term, ensure that other expected invariants on weights are ensured.

7. Incorrect MIN_SINT_32 constant

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK2-7
Target: `lisk-sdk/elements/lisk-validator/src/constants.ts`	

Description
The `MIN_SINT_32` constant, used by Lisk's schema validator, is incorrect: its value is -2147483647, but should be -2147483648, since this is the minimum possible 32-bit signed integer.

Exploit Scenario
A legitimate user tries to use the value -2147483648 in a `sint32` field of a message, but is unable to because the schema validator rejects the message.

Recommendations
Change the value of the `MIN_SINT_32` constant to -2147483648.

8. Event errors are reverted

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-LISK2-8
Target: `lisk-sdk/framework/src/modules/nft/events/{destroy.ts,lock.ts}`	

Description

The `DestroyEvent` and the `LockEvent` classes' error methods do not call the `BaseEvent` class's `add` method with the `noRevert` argument set to `true`. The `noRevert` argument is used to prevent reverting specific events even when command execution fails; it is useful to emit error events that describe the error that caused the execution failure. Since the `DestroyEvent` and the `LockEvent` classes' error methods do not pass this argument to the `add` function, as shown in figures 8.1 and 8.2, and its default value is `false`, the error events—emitted when a transaction is about to fail—will never be seen by the user.

```
public error(ctx: EventQueuer, data: DestroyEventData, result: NftErrorEventResult):
void {
        this.add(ctx, { ...data, result }, [data.address, data.nftID]);
}
```

Figure 8.1: The error method of the DestroyEvent class (lisk-sdk/framework/src/modules/nft/events/destroy.ts#56–58)

```
public error(ctx: EventQueuer, data: LockEventData, result: NftErrorEventResult) {
        this.add(ctx, { ...data, result }, [Buffer.from(data.module), data.nftID]);
}
```

Figure 8.2: The `error` method of the `LockEvent` class (lisk-sdk/framework/src/modules/nft/events/lock.ts#63–65)

We found this issue with the Semgrep rule `error_event_is_not_revert`, which we provided in the first phase of this audit.

```
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 8.3: The `error_event_is_not_revert` Semgrep rule

Exploit Scenario 1

An attacker has an attack against a custom side chain that involves brute forcing the `destroy` method of the NFT module. No error event is emitted for these failures. The attack goes unnoticed.

Exploit Scenario 2

A user or blockchain developer tries to destroy an NFT. Their call to the `destroy` method fails but no error event is emitted. The user fails or takes a very long time to debug the root cause of the failure.

Recommendations

Short term, on every `error` method of a subclass of the `BaseEvent` class (specifically for `DestroyEvent` and `LockEvent`), pass the `noRevert` argument set to `true`. This will ensure that error events are not reverted when commands fail.

Long term, to avoid similar errors in the future, integrate the `error_event_is_not_revert` Semgrep rule, as well as every Semgrep rule that we provided in the first phase of the audit, into the CI/CD pipeline. Additionally, consider adding an `error` method to the `BaseEvent` class that is similar to the `add` method, but with the `noRevert` argument defaulting to `true`. This will make it easier for developers to write code without this buggy pattern.

9. Unspecified and poorly named NFT endpoints

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LISK2-9
Target: NFT module	

Description

The `getCollectionIDs`, `collectionExists`, and `isNFTSupported` endpoints of the NFT module exist in the code but are not defined in LIP-0052.

Furthermore, the `getCollectionIDs` and `collectionExists` functions incorrectly perform a check if an NFT is supported. Instead of using the `isNFTSupported` method of the NFT module, these functions use the code shown in figure 9.1.

```
const supportedNFTsStore = this.stores.get(SupportedNFTsStore);
const chainExists = await supportedNFTsStore.has(ctx, chainID);
```

Figure 9.1: `lisk-sdk/framework/src/modules/nft/endpoint.ts#180–182`

This check does not take into account the existence of the `ALL_SUPPORTED_NFTS_KEY` key in the `SupportedNFTsStore`. Also, `getCollectionIDs` returns the same value if there are no supported collections or if all collections are supported for a given chain.

Finally, the function's names are not clear. Given these unclear names and nonexistent specification, we are not entirely sure of the functions' intended functionality, but alternative names could be `getSupportedCollectionIDs` and `isCollectionIDSupported`.

Recommendations

Short term, specify the functions described above in the LIP, rename them with a clearer name, and fix their functionality accordingly with the developed specification.

Long term, create a process to limit adding code to modules without it being specified in a LIP.

10. The removeSupportAllNFTs function may not remove support for all NFTs

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK2-10
Target: `lisk-sdk/framework/src/modules/nft/method.ts`	

Description

The `removeSupportAllNFTs` method of the NFT module may not remove support of all NFTs if the `ALL_SUPPORTED_NFTS_KEY` is present. The `removeSupportAllNFTs` method (figure 10.1) removes all keys in the `SupportedNFTsStore` that are returned by the `supportedNFTsStore.getAll` method (figure 10.2). However, the `getAll` function will never return the `ALL_SUPPORTED_NFTS_KEY` key because it iterates over all keys of size `LENGTH_CHAIN_ID` but `ALL_SUPPORTED_NFTS_KEY` is the empty string (i.e., does not have a length of `LENGTH_CHAIN_ID`).

```
public async removeSupportAllNFTs(methodContext: MethodContext): Promise<void> {
	const supportedNFTsStore = this.stores.get(SupportedNFTsStore);

	const allSupportedNFTs = await supportedNFTsStore.getAll(methodContext);

	for (const { key } of allSupportedNFTs) {
		await supportedNFTsStore.del(methodContext, key);
	}

	this.events.get(AllNFTsSupportRemovedEvent).log(methodContext);
}
```

Figure 10.1: lisk-sdk/framework/src/modules/nft/method.ts#709–719

```
public async getAll(
	context: ImmutableStoreGetter,
): Promise<{ key: Buffer; value: SupportedNFTsStoreData }[]> {
	return this.iterate(context, {
		gte: Buffer.alloc(LENGTH_CHAIN_ID, 0),
		lte: Buffer.alloc(LENGTH_CHAIN_ID, 255),
	});
}
```

Figure 10.2: lisk-sdk/framework/src/modules/nft/stores/supported_nfts.ts#63–70

We used the test from figure 10.3 to confirm the issue described above.

```
	it('should remove all existing entries even if the ALL_SUPPORTED_NFTS_KEY
```

```
entry exists', async () => {
                    await supportedNFTsStore.save(methodContext,
ALL_SUPPORTED_NFTS_KEY, {
                            supportedCollectionIDArray: [],
                    });

            await
expect(method.removeSupportAllNFTs(methodContext)).resolves.toBeUndefined();

            await expect(supportedNFTsStore.has(methodContext,
ALL_SUPPORTED_NFTS_KEY)).resolves.toBeFalse();

            checkEventResult(methodContext.eventQueue, 1,
AllNFTsSupportRemovedEvent, 0, {}, null);
        });
```

Figure 10.3: Test that fails because the `removeSupportAllNFTs` does not delete the `ALL_SUPPORTED_NFTS_KEY` key. Highlighted in red is the check that fails.

Exploit Scenario

A module relies on the `removeSupportAllNFTs` function for removing support for all NFTs under specific conditions. The call to `removeSupportAllNFTs` fails to remove support for all NFTs. An attacker exploits this fact to break the module's assumptions and exploit the sidechain.

Recommendations

Short term, in the `removeSupportAllNFTs` function, also remove the `ALL_SUPPORTED_NFTS_KEY` key. Add the test in figure 10.3 to the existing test suite.

Long term, write more complex tests for `removeSupportAllNFTs` and similar functions. Edge cases and magic values such as `ALL_SUPPORTED_NFTS_KEY` should be thoroughly tested.

11. Bug in removeSupportAllNFTs tests

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-LISK2-11
Target: `lisk-sdk/framework/test/unit/modules/nft/method.spec.ts`	

Description

The test for the `removeSupportAllNFTs` function incorrectly saves a random wrong value to the `supportedNFTsStore` store.

The test should work by: 1) creating a random `chainID`, 2) saving that `chainID` to the `supportedNFTsStore` store, 3) removing the support for that `chainID`, and 4) checking that the `supportedNFTsStore` no longer contains the `chainID` key. The bug is in step 2, where the code is just saving a random chain ID instead of using the chainID variable (highlighted in red in figure 11.1).

```
const chainID = utils.getRandomBytes(LENGTH_CHAIN_ID);

await supportedNFTsStore.save(methodContext, utils.getRandomBytes(LENGTH_CHAIN_ID),
{
       supportedCollectionIDArray: [],
});

await expect(method.removeSupportAllNFTs(methodContext)).resolves.toBeUndefined();

await expect(supportedNFTsStore.has(methodContext, chainID)).resolves.toBeFalse();
```

Figure 11.1:
lisk-sdk/framework/test/unit/modules/nft/method.spec.ts#1183–1191

Furthermore, by removing the call to `supportedNFTsStore.save` altogether, the test still passes, which is undesirable. The code should first check that the NFT is supported, remove support for all NFTs, and check if the NFT is no longer supported.

Recommendations

Short term, fix the test described above by replacing the call to `supportedNFTsStore.save`'s `utils.getRandomBytes(LENGTH_CHAIN_ID)` argument with the `chainID` variable. Also, add a check in the test that ensures that the NFT is supported before removing it.

Long term, use a tool such as necessist to find other issues in tests.

12. Bounced NFT transfers may cause events to contain incorrect data

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-LISK2-12
Target: `lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts`	

Description

When an error occurs on a cross-chain NFT transfer and the status of the CCM is not `CCM_STATUS_CODE_OK`, the `recipientAddress` variable is set to the value of the `senderAddress` variable in two locations: here and here. The `senderAddress` variable is not updated.

Then, at the end of the function, a `CcmTransferEvent` event is emitted with the `senderAddress` and `recipientAddress` values (figure 12.1), which, as explained above, will always have the same value on status different from `CCM_STATUS_CODE_OK`. The same problem exists in the token module.

```
this.events.get(CcmTransferEvent).log(context, {
        senderAddress,
        recipientAddress,
        nftID,
});
```

Figure 12.1:
lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts#157–161

Another problem is the lack of sending and receiving `chainID` data in the event. While the token module includes the receiving `chainID`, the NFT module includes neither (as shown in figure 12.1).

Exploit Scenario

A developer is tracing where an NFT was transferred to and from to audit an ongoing attack. In the logs, the developer sees that the NFT was transferred from account A to account B and then, because an error occurred, the log will show a transfer from account A to account A. In both cases, the receiving and sending `chainID`s are not included in the event data. This confuses the developer and slows down the auditing process.

Recommendations

Short term, in both the NFT and token module's `CcmTransferEvent` event, update the `senderAddress` to correctly show the sender address of the NFT or token amount. Consider if the event data should also include the sending and receiving `chainID`s, and

update both modules accordingly. This will make the `CcmTransferEvent` events emitted by the NFT and Token modules more expressive and consistent with each other.

Long term, review the events emitted in all other commands. Ensure that they are consistent with similar commands and that they are sufficient to trace the ownership of assets and other required properties.

13. An NFT's attributesArray can have duplicate modules

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LISK2-13
Target: NFT module	

Description

The NFT module defines the `createNFTEntry` function, which is responsible for creating a new NFT and ensuring that the NFT's `attributesArray` array does not have duplicate modules, as highlighted in figure 13.1.

```
public async createNFTEntry(
	methodContext: MethodContext,
	address: Buffer,
	nftID: Buffer,
	attributesArray: NFTAttributes[],
): Promise<void> {
	const moduleNames = [];
	for (const item of attributesArray) {
		moduleNames.push(item.module);
	}

	if (new Set(moduleNames).size !== attributesArray.length) {
		throw new Error('Invalid attributes array provided');
	}

	const nftStore = this.stores.get(NFTStore);
	await nftStore.save(methodContext, nftID, {
		owner: address,
		attributesArray,
	});
}
```

Figure 13.1: lisk-sdk/framework/src/modules/nft/internal_method.ts#63–83

The `createNFTEntry` function is never called; instead, the `nftStore.save` method is called directly in many locations. In some locations, such as in the `create` method, the `attributeArray` uniqueness check is copy-pasted, while in others this uniqueness property is not enforced.

In particular, two places may result in the NFT having duplicate module attributes:

- When a foreign NFT is received (lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts#L142-L145)

- When a foreign NFT is bounced (lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts#L149-L152)

In the `recover` function and cross-chain transfers to the NFT's native chain, this problem also exists but is not exploitable because the incoming NFT attributes are dropped by the current implementation of `getNewAttributes`:

- lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts#L116
- lisk-sdk/framework/src/modules/nft/method.ts#L977

The `createUserEntry` function is also not used here and here, and the `createEscrowEntry` function is not used here.

Exploit Scenario

An attacker sends an NFT with duplicate attributes from chain A (the NFT's native chain) to chain B. A custom module of chain B performs validation of the NFT's attributes by iterating over the attributes array until it finds the first instance corresponding to module X (e.g., using the array's `find` method) and then doing the check. Later, the same custom module gets the attributes for module X in a different way that returns the last instance on the attributes array instead of the first (e.g., by iterating over the whole array and adding the values to a map for faster access). Only the first instance is checked, breaking the guarantees that the module has for the attributes and potentially leading to a security problem.

Recommendations

Short term, in every location where an NFT is created or updated, use a purposely built function (e.g., the `createNFTEntry` function) to create the NFT and ensure the correctness of its properties. Avoid calling `nftStore.save` and similar methods in other code locations.

Long term, in general, avoid modifying the store's state directly, especially when specific properties need to be enforced. Instead, create purposely built functions that enforce the properties in a single centralized location that is easy to audit and avoids code repetition. See figure 4.4 from finding TOB-LISK2-4 to see what the final result could look like.

14. Bounced messages may be abused to bypass NFT fees

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK2-14
Target: `lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts`	

Description

On CCMs with a status different from `CCM_STATUS_CODE_OK` (figure 14.1 highlighted in yellow), the NFT creation does not incur a fee.

```
if (status === CCM_STATUS_CODE_OK) {
	this._feeMethod.payFee(getMethodContext(), BigInt(FEE_CREATE_NFT));
	await nftStore.save(getMethodContext(), nftID, {
		owner: recipientAddress,
		attributesArray: receivedAttributes as NFTAttributes[],
	});
	await this._internalMethod.createUserEntry(ctx, recipientAddress, nftID);
} else {
	recipientAddress = senderAddress;
	await nftStore.save(getMethodContext(), nftID, {
		owner: recipientAddress,
		attributesArray: receivedAttributes as NFTAttributes[],
	});
	await this._internalMethod.createUserEntry(ctx, recipientAddress, nftID);
}
```

Figure 14.1: lisk-sdk/framework/src/modules/nft/cc_commands/cc_transfer.ts#140–154

This code works this way because messages with a status different from `CCM_STATUS_CODE_OK` are expected to be a bounced CCM, i.e., a CCM where the NFT is being recreated after a previous cross-chain transfer failed. However, there is nothing preventing a malicious sidechain from initiating a CCM with any status, enabling them to transfer NFTs cross-chain without paying the fee.

Exploit Scenario

Chain A sends an NFT (native to Chain A) to chain B, setting the message status to a value different from `CCM_STATUS_CODE_OK` (e.g., `CCMStatusCode.MODULE_NOT_SUPPORTED`). Chain B receives the foreign NFT and creates the NFT entry without the fee payment.

Recommendations

Short term, enforce a fee payment even on bounced messages. If this is not possible or wanted, modify the interoperability module to guarantee that the mainchain enforces that bounced messages cannot be initiated by malicious side chains.

Long term, review other modules' handling of bounced messages to ensure similar problems do not exist.

15. NFT recover function may crash

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK2-15
Target: `lisk-sdk/framework/src/modules/nft/method.ts`	

Description

The `recover` function of the NFT module does not verify that the NFT to be recovered exists in its `NFTStore`. This causes the `nftStore.get` call highlighted in figure 15.1 to crash.

```
const nftData = await nftStore.get(methodContext, nftID);
if (!nftData.owner.equals(terminatedChainID)) {
```

Figure 15.1: `lisk-sdk/framework/src/modules/nft/method.ts#940–941`

Exploit Scenario

A malicious sidechain sends a fake recovered NFT that does not exist on its native chain. The native chain's `recover` function crashes.

Recommendations

Short term, add a check to ensure that the NFT exists before accessing its data. This will ensure the `recover` function throws an exception explicitly instead of crashing unexpectedly.

Long term, write fuzzing harnesses that send arbitrary objects (that are valid according to given schema) to the `recover` method to simulate the behavior of malicious sidechains.

16. NFT recover function does properly validate NFT attributes array

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-LISK2-16
Target: `lisk-sdk/framework/src/modules/nft/method.ts`	

Description

The `recover` function of the NFT module does not call `validator.validate` on the `storeValue` variable after decoding the value (figure 16.1).

```
try {
	decodedValue = codec.decode<NFTStoreData>(nftStoreSchema, storeValue);
} catch (error) {
	isDecodable = false;
}
```

Figure 16.1: lisk-sdk/framework/src/modules/nft/method.ts#902–908

As such, the NFT attributes' module names are not validated for length or pattern with the `nftStoreSchema`, which is shown in figure 16.2.

```
export const nftStoreSchema = {
	$id: '/nft/store/nft',
	type: 'object',
	required: ['owner', 'attributesArray'],
	properties: {
		owner: {
			dataType: 'bytes',
			fieldNumber: 1,
		},
		attributesArray: {
			type: 'array',
			fieldNumber: 2,
			items: {
				type: 'object',
				required: ['module', 'attributes'],
				properties: {
					module: {
						dataType: 'string',
						minLength: MIN_LENGTH_MODULE_NAME,
						maxLength: MAX_LENGTH_MODULE_NAME,
						pattern: '^[a-zA-Z0-9]*$',
						fieldNumber: 1,
					},
```

```
                                attributes: {
                                        dataType: 'bytes',
                                        fieldNumber: 2,
                                },
                            },
                        },
                    },
                },
        },
};
```

Figure 16.2: `lisk-sdk/framework/src/modules/nft/stores/nft.ts#28–59`

The following test can be used to confirm that the call to `codec.decode` does not validate a string's length.

```
describe('length_validation', () => {
    const schema_with_max_length = {
        $id: '/lisk/decode/with_max_length',
        type: 'object',
        properties: {
            foo: {
                dataType: 'string',
                fieldNumber: 1,
                minLength: 2,
                maxLength: 2,
            },
        },
    };

    it('should work with a valid length of 2', () => {
        const res = codec.encode(schema_with_max_length, {foo: "12"});
        expect(codec.decode(schema_with_max_length, res)).toEqual({foo: "12"});
    });

    it('should not work with length different than 2', () => {
        const res = codec.encode(schema_with_max_length, {foo: "12345"});
        expect(codec.decode(schema_with_max_length, res)).toThrow(); // This does
not throw
    });
});
```

Figure 16.3: Test that confirms that calls to `codec.decode` do not validate a string's length

This finding is informational because the attributes are not used unless a custom module implements the `getNewAttributes` function.

Exploit Scenario

An attacker recovers an NFT from a chain they own to a sidechain that reimplements the `getNewAttributes` function to use the received NFT attributes. The `NFTStore` store saves data that is inconsistent with the schema.

Recommendations

Short term, have the code call the `validator.validate` function on the decoded `NFTStoreData` data.

Long term, review every other module's `recover` methods for the same issue.

17. The codec.encode function encodes larger than expected integers	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LISK2-17
Target: Codec class	

Description

The encode function of the Codec class successfully encodes integers larger than expected. This issue can be replicated with the test shown in figure 17.1, in which a value of 2^{100} is successfully encoded as a `uint64` (while the maximum value should be 2^{64}-1).

```
describe('uint_validation', () => {
    const schema_uint64 = {
        $id: 'test/uint_validation',
        type: 'object',
        required: ['amount'],
        properties: {
            amount: {
                dataType: 'uint64',
                fieldNumber: 1,
            },
        },
    };

    it('uint64 encoding with larger than expected values', () => {
        expect(codec.encode(schema_uint64, {amount:
BigInt(2)**BigInt(100)})).toThrow(); // This does not throw
    });
});
```

Figure 17.1: Test that shows the encode function of the Codec class successfully encoding integers larger than expected

Calling the decode method of the Codec class on this encoded data fails, breaking the round-trip property of the encode and decode functions.

Exploit Scenario

A user successfully stores a value in a module's store that is larger than the maximum integer. The user attempts to use the module again but is unable to because the call to decode fails.

Recommendations

Short term, add checks in the code to prevent integers larger than the maximum value or smaller than the minimum value cannot be encoded or decoded.

Long term, extend the test suite for integer encoding and decoding to detect these and similar issues. Always tests the corner cases, such as the maximum and minimum integer boundaries.

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
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 Recommendations

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, may become easily exploitable in future releases, or may decrease code readability. We recommend fixing the following issues:

NFT Module

1. **Incorrect regex in LIP.** The LIP's userStoreSchema schema specification defines the pattern for the `lockingModule` as `^[a-zA-Z0-9]*$]`. The last `]` is a typo and should not be present. The pattern should be `^[a-zA-Z0-9]*$`. The corresponding implementation is correct.

2. **Constants are defined but hard-coded versions are used**. In `module.ts#L124-L126`, `'nft'` is used instead of `MODULE_NAME_NFT`.

3. **Unnecessary calls to `validator.validate`.** In the NFT module's transfer and transfer cross chain commands, the code calls `validator.validate` to validate the commands' parameters. Calling `validator.validate` on a command's `verify` function is unnecessary because this check is already done in the state machine logic.

4. **Use of deprecated `Buffer.slice` function.** The codes uses `Buffer.slice`—a deprecated function—in `method.ts#L77`, `method.ts#L196`, and `method.ts#L289`. Instead, the code should use `Buffer.subarray`.

5. **Check missing from LIP.** The `getChainID` function implementation has a check for the `nftID` length, which is not in the corresponding LIP specification. The LIP should be updated to include this check.

6. **Bad pseudocode notation in the LIP.** The LIP contains inconsistent notation in a couple of locations:

    a. The specification of the `getCollectionID` function has a poor notation for sub-arraying the `nftID` variable. The LIP uses `nftID[`LENGTH_CHAIN_ID`:(`LENGTH_CHAIN_ID` + `LENGTH_COLLECTION_ID`)]`. We recommend removing the ` characters, resulting in this notation: `nftID[LENGTH_CHAIN_ID:(LENGTH_CHAIN_ID + LENGTH_COLLECTION_ID)]`.

    b. The specification of the `getAttributes` function has a return statement with unnecessary parenthesis. The LIP's return statement is represented as `return(entry['attributes'])`, but the parenthesis are unnecessary.

Instead, to keep the notation consistent with other returns in the LIP, the pseudo-code should be `return entry['attributes']`.

7. **Lack of constant definition.** The code calculates the length of the `index` portion of an `nftID` in two locations (`method.ts#L276`, and `method.ts#L315`) with the code `indexLength = LENGTH_NFT_ID - LENGTH_CHAIN_ID - LENGTH_COLLECTION_ID`. Instead, this value should be defined as a constant like `LENGTH_CHAIN_ID`, and `LENGTH_COLLECTION_ID`.

8. **Inconsistent error handling in command `verify` functions.** For example, the NFT module throws an exception on errors, while the token module returns a `VerificationResult` with a `VerifyStatus` of `TransactionVerifyResult.INVALID`. In both cases, the state machine handles the errors similarly (only altering the debug log that is created); however, the error handling for all command's `verify` functions should be consistent.

9. **Inconsistent event data.** The `CreateEvent` event data includes the `collectionID` field. This field is not present in the `lock`, `unlock`, or `destroy` events, which include only the `nftID` (from which the NFT's collection ID can be extracted). All of these events that modify the state of NFTs should be consistent in the data they include, either including the `collectionID` field in all of them or in none of them.

10. **Check in LIP-0052 but not in the implementation.** We found the following instances where checks are not defined in the specification but are present in the implementation:

    a. In the `removeSupportAllNFTsFromCollection` function, the `chainID.equals(this._config.ownChainID)` check is not defined in the LIP specification.

    b. In the `supportAllNFTs` function, the check that tests if the ALL_SUPPORTED_NFTS_KEY key is present in the SupportedNFTsStore is missing from the LIP.

PoA Module

1. **Misspelled, unused constant**. In `poa/constants.ts`, both `LENGTH_PROOF_OF_POSSESSION` and `LENGTH_PROOF_OF_POSESSION` are defined, both equaling 96. `LENGTH_PROOF_OF_POSESSION` is never used.

2. **Redundant, unused type**. In `poa/stores/snapshot.ts`, the `Validator` type is defined but never used. It is identical to the `ActiveValidator` type, imported from `poa/types.ts`. `ActiveValidator` is used later in `poa/stores/snapshot.ts`.

3. **Hard-coded SUBSTORE_PREFIX constants**. The LIP specifying the PoA module specifies four SUBSTORE_PREFIX constants. These constants are hard coded in the implementation (in `poa/module.ts`, in the constructor of `PoAModule`), rather than being defined as constants in `poa/constants.ts`.

4. **Constants are defined but hard-coded versions are used**:

    a. In `poa/module.ts`, line 88, `'poa'` is used instead of `MODULE_NAME_POA`.

    b. In `poa/module.ts`, line 216, `/^[a-z0-9!@$&_.]+$/g` is used instead of `POA_VALIDATOR_NAME_REGEX`. While the hard-coded value is slightly different from the constant (the hard-coded value includes a `g` for global at the end, while the constant does not), the effect will be the same.

    c. In `poa/endpoint.ts`, on lines 73 and 74, `20` is used instead of `NUM_BYTES_ADDRESS`.

5. **Inconsistent use of Buffer.from**. Some code locations use `Buffer.from(str, 'utf-8')` while other locations use `Buffer.from(str)`, leaving out the `'utf-8'` argument. The effect is the same, since `'utf-8'` is the default value of that argument.

6. **`RegisterAuthorityParams` fields are ordered differently than in the LIP**. The `generatorKey` and `proofOfPossession` fields are swapped. This could potentially cause schema matching problems if JSON serialization is done in the wrong order. However, we did not find any security problems related to this issue.

7. **Unnecessary calls to `validator.validate`.** In the PoA module's register authority and update generator key commands, the code calls `validator.validate` to validate the commands parameters. Calling `validator.validate` on a command's `verify` function is unnecessary because this check is already done in the state machine logic.

8. **Array ordering is checked manually when a helper function could be used.** In the `PoAModule.initGenesisState` function, there is a check for whether the `validatorAddresses` list is sorted, by comparing each of its elements against the corresponding element in `[...validatorAddresses].sort()`. Instead, the `objectUtils.isBufferArrayOrdered` helper function should be used. The same applies to the check done later in the function on the `activeValidators` list: `objectUtils.isBufferArrayOrdered(activeValidatorAddresses)` should be used.

9. **The `shuffleValidatorList` function is defined both in PoA and PoS.** Both do essentially the same thing, but are implemented in slightly different ways. Ideally, they should use a common implementation.

10. **In LIP-0047 for PoA, `genesisPoAStoreSchema.snapshotSubstore` is missing a field number.** This problem occurs only in the LIP, and not in the implementation.

11. **The `getAllValidatorsResponseSchema` schema specifies that weights should have a `'uint64'` format, while the implementation may assign them to be blank strings.** This is not a security issue, since there is no formal definition of the `'uint64'` format anyway (see the fix review for TOB-LISK-52 from the first phase of the audit), and since the response is never validated against the schema.

12. **Unnecessary store lookup.** In the `PoAEndpoint.getAllValidators` function, `validatorStore.iterate` is used to get a list of validator `(address, name)` pairs. However, later on, the addresses in the returned list are again looked up using `validatorStore.get(context, data.key)`, when `data.value` could be used instead.

13. **Unnecessary call to `getAddressFromPublicKey`.** The register authority command uses `address.getAddressFromPublicKey(context.transaction.senderPublicKey)` in its `verify` function to get the transaction's sender address, when it could instead use `context.transaction.senderAddress`.

D. 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 NFT and PoA modules using the following public rulesets:

- `p/javascript`
- `p/r2c`
- `p/r2c-security-audit`
- `p/r2c-best-practices`
- `p/nodejs`
- `p/nodejsscan`

We also used the custom rules provided in the previous part of this audit:

- `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`
- `schema_property_element_without_field_number`
- `schema_with_duplicate_field_number`
- `schema_with_field_not_required`
- `verify_without_schema_verify`
- `schema_with_required_that_is_not_a_property`
- `schema_int_format_with_integer_type`
- `schema_hardcoded_pattern`

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 D.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 D.1: Commands used to run CodeQL

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 October 18 to October 25, 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, the Lisk Foundation team has resolved all issues described in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Impossible to send cross-chain NFT transfers in certain configurations	**Resolved**
2	NFT module has repeated transfer functionality	**Resolved**
3	NFT lock method does not check if module is NFT_NOT_LOCKED	**Resolved**
4	NFT methods API could be improved	**Resolved**
5	Checks done manually instead of using schema validation	**Resolved**
6	Update authority command allows validators to have 0 weight	**Resolved**
7	Incorrect MIN_SINT_32 constant	**Resolved**
8	Event errors are reverted	**Resolved**
9	Unspecified and poorly named NFT endpoints	**Resolved**

10	The removeSupportAllNFTs function may not remove support for all NFTs	Resolved
11	Bug in removeSupportAllNFTs tests	Resolved
12	Bounced NFT transfers may cause events to contain incorrect data	Resolved
13	An NFT's attributesArray can have duplicate modules	Resolved
14	Bounced messages may be abused to bypass NFT fees	Resolved
15	NFT recover function may crash	Resolved
16	NFT recover function does properly validate NFT attributes array	Resolved
17	The codec.encode function encodes larger than expected integers	Resolved

Detailed Fix Review Results

TOB-LISK2-1: Impossible to send cross-chain NFT transfers in certain configurations
Resolved in PR #8963.

TOB-LISK2-2: NFT module has repeated transfer functionality
Resolved in PR #9005.

TOB-LISK2-3: NFT lock method does not check if module is NFT_NOT_LOCKED
Resolved in PR #8992.

TOB-LISK2-4: NFT methods API could be improved
Resolved in PRs #9034 and #9005.

TOB-LISK2-5: Checks done manually instead of using schema validation
Resolved in PR #8967 and #8953.

TOB-LISK2-6: Update authority command allows validators to have 0 weight
Resolved in PR #8953.

TOB-LISK2-7: Incorrect MIN_SINT_32 constant
Resolved in PR #8976.

TOB-LISK2-8: Event errors are reverted
Resolved in PR #8978.

TOB-LISK2-9: Unspecified and poorly named NFT endpoints
Resolved in PR #9036.

TOB-LISK2-10: The removeSupportAllNFTs function may not remove support for all NFTs
Resolved in PR #9017.

TOB-LISK2-11: Bug in removeSupportAllNFTs tests
Resolved in PR #9018.

TOB-LISK2-12: Bounced NFT transfers may cause events to contain incorrect data
Resolved in PR #9011. We encourage the Lisk team to apply the same fix to the token module. Currently, the token module logs `receivingChainID` but not `sendingChainID`.

TOB-LISK2-13: An NFT's attributesArray can have duplicate modules
Resolved in PR #9014.

TOB-LISK2-14: Bounced messages may be abused to bypass NFT fees
Resolved in PR #9011.

TOB-LISK2-15: NFT recover function may crash
Resolved in PR #9028.

TOB-LISK2-16: NFT recover function does properly validate NFT attributes array
Resolved in PR #9000.

TOB-LISK2-17: The codec.encode function encodes larger than expected integers
Resolved in PR #9010.

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.