Audit Report

Fairblock fairyring

v1.0

October 2, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Fairblock Inc to perform a security audit of the fairyring Cosmos SDK chain.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/fairblock/fairyring
Commit	`46597175c7bfc588e732405cd518edfc717c8a17`
Scope	All Cosmos SDK code in the `app`, `cmd`, and `x` directories was in scope.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The fairyring chain enables the transmission of encrypted transactions through the utilization of a public key. These transactions are automatically decrypted and executed at the specified height. To ensure the effectiveness of this process, a unique private key is generated for each block, eliminating the possibility of front-running. Additionally, once the decryption key becomes accessible, the encrypted transactions are prioritized for execution over any other transactions in the mempool.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Low-Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	-
Test coverage	Low	Unit tests are failing and are not up-to-date. Integration tests are not extensive.

Summary of Findings

No	Description	Severity	Status
1	Attackers can register unbonded validators in the `keyshare` module to submit malicious key shares, censor, or DoS the chain	Critical	Resolved
2	Attackers can register validators in the `staking` module to DoS the chain	Critical	Resolved
3	Validators can censor the execution of encrypted transactions at a particular block height without being punished	Critical	Resolved
4	Encrypted transaction execution does not charge gas	Critical	Resolved
5	Attackers can overwrite legitimate `AggregateKeyShares` making the chain unable to execute transactions	Critical	Resolved
6	Attackers can submit a large number of `MsgSubmitEncryptedTx` targeting the same block height to DoS the chain	Critical	Resolved
7	Attackers can submit a transaction with a large number of `MsgCreateAggregatedKeyShare` messages to DoS the chain	Critical	Resolved
8	Logged/emitted keys combined with ability to aggregate keys for future block heights allows decryption of transactions	Critical	Resolved
9	If `MsgCreateAggregatedKeyShare` is not in the first 30 transactions in the mempool, blocks are executed without the decryption key	Critical	Resolved
10	Insufficient validation of `PepNonce` can lead to transaction replay attack	Critical	Resolved
11	Attackers could overwrite the active public key via IBC, leading to the inability of executing encrypted transactions	Critical	Resolved
12	Insufficient verification of submitted key shares could lead to the inability to aggregate the key	Critical	Resolved
13	`TrustedAddresses` are a single point of failure	Minor	Acknowledged

14	Missing validation in `keyshare` module's `Params`	Minor	Resolved
15	Missing `MsgSendKeyshare` input fields validation	Minor	Resolved
16	Missing validations of `GenesisState`	Minor	Resolved
17	`Amino` codec must be registered to support users with hardware devices like Ledger	Minor	Resolved
18	Parsing query command flags are ignored	Minor	Resolved
19	`MsgSendKeyshare` transaction silently fails	Minor	Resolved
20	Use of magic numbers decreases maintainability	Informational	Resolved
21	Inefficient execution of `GetAllValidators`	Informational	Resolved
22	Open issues in a dependency	Informational	Acknowledged
23	Missing usage description for transaction and query CLI commands	Informational	Resolved
24	Duplicated `ValidateBasic` invocation in CLI	Informational	Resolved
25	Non-standard management of external functions	Informational	Resolved
26	Miscellaneous code quality comments	Informational	Resolved

Detailed Findings

1. Attackers can register unbonded validators in the `keyshare` module to submit malicious key shares, censor, or DoS the chain

Severity: Critical

The `MsgRegisterValidator` handler in `x/keyshare/keeper/msg_server_register_validator.go:12-38` enables the registration of a new validator in the `keyshare` module without conducting a validation check to verify if the provided validator is bonded.

This allows an attacker to create and register a significant number of unbonded validators within the `keyshare` module. Consequently, this vulnerability enables the attacker to potentially DoS the chain, engage in chain censorship, or submit malicious key shares.

To execute chain censorship, attackers can register an extensive quantity of unbonded validators. Since the creation of an aggregate key necessitates reaching a threshold that depends on the number of validators, a substantial increase in the number of validators can elevate the threshold. This elevated threshold can render it unfeasible for legitimate validators to create the aggregated key.

Another possible attack scenario involves the submission of malicious key shares, where a single entity possesses all of them, front-running legitimate validators. In this scenario, the attacker can generate the aggregated key and decrypt transactions.

Furthermore, validators registered in the `keyshare` module are iterated over during the `SendKeyshare` transaction handling without imposing any gas limitations in `x/keyshare/keeper/msg_server_send_keyshare.go:81-87`, `133-148`, and `151`. Consequently, the execution of this message can place excessive strain on the node's resources, potentially resulting in a slowdown of the chain or, in more severe circumstances, a complete halt.

Recommendation

We recommend not allowing unbonded validators to register in the `keyshare` module.

Status: Resolved

2. Attackers can register validators in the `staking` module to DoS the chain

Severity: Critical

During the execution of the `BeginBlock` function, specifically in `x/keyshare/module.go:162`, an iteration is performed over all the validators registered in the `staking` module, regardless of their bonded status.

Since the `staking` module allows anyone to register a validator, an attacker could register a substantial number of them, resulting in the `BeginBlock` function iterating through all of these validators.

As a consequence, the execution of the `BeginBlock` function will consume more time and resources than anticipated, potentially leading to a slowdown of the chain or, in severe cases, a complete halt.

Furthermore, it is important to note that a significant quantity of validators within the `staking` module can contribute to increased resource demands during the execution of the `RegisterValidator` message handler.

Specifically, in `x/keyshare/keeper/msg_server_register_validator.go:25`, the handler iterates through all registered validators without imposing any gas limitations. Consequently, the execution of this message can place excessive strain on the node's resources, potentially resulting in a slowdown of the chain or, in more severe circumstances, a complete halt.

Recommendation

We recommend not iterating through all registered validators but using the `GetLastValidators` function to get bonded ones and remove all the others from the store.

Status: Resolved

3. Validators can censor the execution of encrypted transactions at a particular block height without being punished

Severity: Critical

Validators can censor encrypted transactions at a particular block height by not submitting their key shares.

There is no mechanism in place to punish idle validators who do not submit key shares. This would affect the reliability of the transaction execution since there is no guarantee for users that their transaction will be executed.

Additionally, transactions that are not executed at the target height are discarded and there is no fallback mechanism in place to try to execute them in the next block.

Recommendation

We recommend implementing a slashing mechanism for idle validators.

Status: Resolved

4. Encrypted transaction execution does not charge gas

Severity: Critical

The execution of submitted encrypted transactions occurs within the `BeginBlock` function at the designated target height, as defined in `x/pep/module.go:513`.

However, due to the nature of these transactions being executed directly in the `BeginBlock` function without user initiation, there is currently no mechanism in place to invoke the `AnteHandler` and charge the user for gas consumption. This vulnerability can potentially be exploited by attackers to carry out a variety of attacks.

For instance, attackers could attempt to launch a Denial-of-Service (DoS) attack on the chain, saturate the block space by flooding it with transactions, or execute IBC transactions with the intent of targeting the funds of relayers.

Recommendation

We recommend charging encrypted transaction execution gas to users.

Status: Resolved

5. Attackers can overwrite legitimate `AggregateKeyShares` making the chain unable to execute transactions

Severity: Critical

The `CreateAggregatedKeyShare` function, defined in `x/pep/keeper/msg_server_aggregated_key_share.go:10` enables anyone to submit the `AggregatedKeyShare` to for a particular block height.

Since the validation of the submitted key is executed only in the `BeginBlock` function, attackers could be able to overwrite the legit key submitted by `FairyPort` with an invalid one. The only condition needed is that their transaction is executed after the legitimate one.

Consequently, the chain will not be able to decrypt and process transactions for that block height.

Recommendation

We recommend validating the submitted `AggregatedKeyShare` during the `MsgCreateAggregatedKeyShare` message handling.

Status: Resolved

6. Attackers can submit a large number of `MsgSubmitEncryptedTx` targeting the same block height to DoS the chain

Severity: Critical

The `SubmitEncryptedTx` function, defined in `x/pep/keeper/msg_server_submit_encrypted_tx.go:12` enables users to submit encrypted transactions that will be executed at the defined `TargetBlockHeight`.

Those transactions are then iterated over, and each one is decrypted and executed at the target block height in the `BeginBlock` function in line `x/pep/module.go:223`.

Since there is no limit to the number of transactions that could be registered to be executed at a certain block height, attackers could submit a large amount of such transactions in order to spam the network. It is worth noting that a transaction can contain multiple messages and that the attacker will only pay for the byte length of the encrypted transaction.

If the `PEP` module is deployed on chains that support CosmWasm, it opens up the possibility of deploying ad-hoc contracts, dispatching a large number of messages with a lightweight transaction, and executing computationally intensive operations with minimal cost.

Consequently, the iteration, decoding, and execution of those transactions can place excessive strain on the node's resources, potentially resulting in a slowdown of the chain or, in more severe circumstances, a complete halt.

Recommendation

The current approach of executing any arbitrary number of transactions at a specific height is not sustainable. Furthermore, accurately estimating the gas cost and complexity of encrypted transactions is not feasible.

We recommend implementing a hard cap on the number of transactions per block and charging the maximum gas limit to users when submitting encrypted transactions. Then, after the transaction execution in the `BeginBlock` function, any unused gas from transaction processing could be refunded to the user.

Status: Resolved

7. Attackers can submit a transaction with a large number of `MsgCreateAggregatedKeyShare` messages to DoS the chain

Severity: Critical

The `ProcessUnconfirmedTxs` function defined in `x/pep/keeper/unconfirmed_txs.go:21` is executed during the `BeginBlock` function and iterates over a set of unconfirmed transactions and their inner messages in order to find `MsgCreateAggregatedKeyShare` messages and process them.

Since there is no limit to the number of `MsgCreateAggregatedKeyShare` messages in the mempool, except the mempool size, and no gas for adding them is charged to the sender because the transactions are not directly executed, an attacker could craft and send a large number of transactions containing a large amount of `MsgCreateAggregatedKeyShare` messages in order to DoS attack the chain.

Consequently, the iteration and execution of those transactions can place excessive strain on the node's resources, potentially resulting in a slowdown of the chain or, in more severe circumstances, a complete halt.

Recommendation

We recommend moving the `AggregatedKeyShare` validation from the `BeginBlock` function to the `MsgCreateAggregatedKeyShare` message handler.

Status: Resolved

8. Logged/emitted keys combined with ability to aggregate keys for future block heights allows decryption of transactions

Severity: Critical

In `x/keyshare/keeper/msg_server_send_keyshare.go:165-174`, after successful creation of the aggregated key, the key is stored, emitted in an event, and logged in the console.

Since it is possible for validators to submit key shares and create the aggregated key for every height, including for blocks in the future, such keys may be unintentionally exposed, allowing anyone to decrypt transactions and predict future chain states.

Users can simply monitor the chain state, listen for emitted events, or check the console of a node they run to retrieve the key.

Recommendation

We recommend not logging or emitting sensitive information and not allowing validators to aggregate and store keys for blocks in the future.

Status: Resolved

9. If `MsgCreateAggregatedKeyShare` is not in the first 30 transactions in the mempool, blocks are executed without the decryption key

Severity: Critical

During the `pep` module's `BeginBlock` function, `MsgCreateAggregatedKeyShare` messages are retrieved from the mempool in order to execute them before other transactions in line `x/pep/module.go:188`. This is needed in order to have the current block decryption key in the store.

However, since the `tmcore.UnconfirmedTxs` function returns only the first 30 transactions in the mempool, there is no guarantee that `MsgCreateAggregatedKeyShare` messages are retrieved and executed before other transactions.

Consequently, the decryption key of the current block may not be processed and all the encrypted transactions with target execution at the current height might be skipped.

Recommendation

We recommend redesigning the flow that facilitates the provision of the aggregate key to the chain without iteration over all transactions from the mempool. `tmcore.UnconfirmedTxs` may contain a significant number of transactions, and iterating through them in the `BeginBlock` function to search for `MsgCreateAggregatedKeyShare` messages results in an unbounded iteration. This could lead to a Denial-of-Service (DoS) or even cause the chain to halt.

Status: Resolved

10. Insufficient validation of `PepNonce` can lead to transaction replay attack

Severity: Critical

In `x/pep/module.go:457-460`, during the processing of encrypted transactions, the `PepNonce` is utilized to prevent replay attacks.

In the current implementation, whether a transaction is accepted or rejected primarily depends on a comparison between a user's nonce and a stored nonce.

Specifically, if the user's nonce is lower than the stored nonce, the transaction is denied. The system operates by pulling the current stored nonce, increasing it by 1, and then setting this incremented nonce minus 1 as the anticipated nonce. Afterward, the system checks whether the user's nonce is less than this anticipated nonce. If it is, the transaction is disallowed.

This mechanism can be exploited as it allows a replay of the first transaction with nonce 1.

Consider this scenario: The initial value of the stored nonce is 0, and a user starts off with a nonce of 1. This transaction is validated. As per a code condition, if the user's nonce is greater than the anticipated nonce, the current nonce is updated to match the user's nonce.

Consequently, the stored nonce becomes 1. Herein lies the possibility of a replay attack due to incorrect calculations. When the user attempts to execute the same transaction for a second time, using a nonce of 1, the anticipated nonce is also 1. According to the system's acceptance criteria, a transaction fails if the user's nonce is less than the anticipated nonce.

However, since 1 isn't less than 1, this condition fails, resulting in the wrongful acceptance of the replayed transaction.

Please refer to Appendix 1 for a detailed dry run of the code, including flow diagrams, to illustrate the scenario where transactions can be replayed.

Recommendation

We recommend handling the specific edge case by setting the `PepNonce` to 1 by default instead of 0 in `x/pep/keeper/pep_nonce.go:60-64`.

Status: Resolved

11. Attackers could overwrite the active public key via IBC, leading to the inability of executing encrypted transactions

Severity: Critical

In `x/pep/keeper/current_keys.go:19`, during the `EndBlock` function, the `PEP` module queries the active public key using the Inter-Blockchain Communication (IBC) protocol.

To construct the IBC packet, it retrieves the source channel by calling the `k.getChannel` function, which retrieves the value stored with a prefix key determined through `KeyPrefix("pep-channel-")`. This value is initially set during the `OnChanOpenConfirm` method of the IBC protocol in `x/pep/module_ibc.go:115`. However, it is important to note that this value can be overwritten whenever a new channel connects.

This presents a potential vulnerability, as a malicious actor can create a fraudulent IBC channel to manipulate the stored value of the active public key in `OnAcknowledgementPacket` in `x/pep/keeper/current_keys.go:147`. A manipulated active public key leads to the inability to decrypt and execute encrypted transactions.

Recommendation

We recommend verifying the source channel information in `OnAcknowledgementPacket` function, specifically the channel ID, against a whitelisted set of channels configured in the system.

Status: Resolved

12. Insufficient verification of submitted key shares could lead to the inability to aggregate the key

Severity: Critical

In the `parseKeyShareCommitment` function, defined in `x/keyshare/keeper/msg_server_send_keyshare.go:35`, all inputs to this method, including `msg.Message`, `msg.Commitment`, `msg.KeyShareIndex`, and `ibeID`, are provided by the user.

The concern is that a malicious validator can generate their own local key share and commitments since the `parseKeyShareCommitment` method used within `send_key_share` does not raise any errors to detect potential manipulations.

Consequently, if a manipulated message is included, and the key share stored, it could lead to the failure of the aggregated key share generation process.

Recommendation

We recommend implementing logic to verify key shares using VSS or other techniques to detect manipulated key shares.

Status: Resolved

13. `TrustedAddresses` are a single point of failure

Severity: Minor

The protocol depends on the participation of third-party actors, denoted as `TrustedAddresses` and stored in `Params`, to regularly update the public key.

However, this process lacks an incentivization mechanism and relies on a closed set of off-chain actors.

Consequently, the operational effectiveness of the chain is entirely dependent on the continual availability of these actors and the accuracy of the information they provide.

This poses a centralization issue which leads to a single point of failure for the chain operation.

Recommendation

We recommend exploring different mechanisms to provide the public key to the chain in a more decentralized way.

Status: Acknowledged

The client states they have already developed a Decentralized Key Generation library that allows validators to generate keys without relying on a centralized trusted service.

This library is planned to be incorporated into the chain following the launch of the centralized version and the establishment of initial traction, thus minimizing unnecessary complexities.

14. Missing `keyshare` module's `Params`

Severity: Minor

During the validation process of the `Params` within the `keyshare` module, specifically in the code segment located in `x/keyshare/types/params.go:84-91`, the `validateTrustedAddresses` function solely verifies that the input slice is a string. Address validation is currently not performed.

Recommendation

We recommend verifying that all the addresses in the provided slice are valid.

Status: Resolved

15. Missing `MsgSendKeyshare` input fields validation

Severity: Minor

During the validation process of the `MsgSendKeyshare`, specifically in `x/keyshare/types/message_send_keyshare.go:44-50`, the `ValidateBasic` function solely verifies that the sender address is valid.

Since the message contains other input data like `message`, `commitment`, and `blockHeight`, they should also be validated before handling the message.

Recommendation

We recommend validating all the `MsgSendKeyshare` input fields in the `ValidateBasic` function.

Status: Resolved

16. Missing validations of `GenesisState`

Severity: Minor

During the validation process of the `pep` module's `GenesisState`, specifically in `x/pep/types/genesis.go:20-54`, the `Validate` function does not check that `PepNonce`'s keys are valid addresses.

Similarly, during the validation of the `keyshare` module's `GenesisState` in `x/keyshare/types/genesis.go`, the `Validate` function does not check if there is a duplicate `Validator` or `ConsAddr` in lines `27-33`.

Recommendation

We recommend validating all the `GenesisState` fields before storing them.

Status: Resolved

17. `Amino` codec must be registered to support users with hardware devices like Ledger

Severity: Minor

In `x/keyshare/types/codec.go:33` and `x/pep/types/codec.go:29`, `Amino` should be used to register all interfaces and concrete types for the `keyshare` and `pep` modules. This is necessary for JSON serialization to support hardware devices like Ledger since these devices do not support proto transaction signing.

Recommendation

We recommend registering all interfaces and concrete types for the `keyshare` and `pep` modules using `Amino` to support JSON serialization.

Status: Resolved

18. Parsing query command flags are ignored

Severity: Minor

Most query CLI commands registered in the `GetQueryCmd` function in `x/keyshare/client/cli/query.go` and `x/pep/client/cli/query.go` use `GetClientContextFromCmd` to get `Context`, which does not read query command flags.

Recommendation

We recommend replacing `GetClientContextFromCmd` with `GetClientQueryContext`.

Status: Resolved

19. `MsgSendKeyshare` transaction silently fails

Severity: Minor

The `SendKeyshare` function, defined in `x/keyshare/keeper/msg_server_send_keyshare.go:20`, silently fails if the sent key share is not valid.

Consequently, a third party that executes this transaction cannot correctly handle a failure, and users are not getting feedback on the execution status.

Recommendation

We recommend adding a field with the execution status to the returned `MsgSendKeyshareResponse` and emitting an event accordingly.

Status: Resolved

20. Use of magic numbers decreases maintainability

Severity: Informational

In `x/keyshare/keeper/msg_server_send_keyshare.go:51` a hard-coded number literal without context or a description is used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Resolved

21. Inefficient execution of `GetAllValidators`

Severity: Informational

In `x/keyshare/keeper/msg_server_register_validator.go:24`, `GetAllValidators` is called in the native `staking` module to obtain all validators. The purpose of obtaining all validators is to verify if any of the validator addresses match the address of the transaction sender. Querying all validators is inefficient though, since there is a `GetValidator` function available in the `staking` module that returns a particular validator. Replacing the `GetAllValidators` function with the `GetValidator` function will clean up the code and reduce resource as well as gas consumption.

Recommendation

We recommend replacing `GetAllValidators` with `GetValidator`.

Status: Resolved

22. Open issues in a dependency

Severity: Informational

An open issue has been identified by Trail of Bits in the `Kilic/bls12-381` library, regarding zero value deserialization due to internal modulo operations.

The mentioned library is utilized in the code and it is crucial to verify that the deserialization issue does not have any adverse effects on the code's functionality and security.

Recommendation

We recommend conducting an impact assessment to evaluate the potential consequences of using this vulnerable dependency in the code base.

Status: Acknowledged

23. Missing usage description for transaction and query CLI commands

Severity: Informational

The transaction and query CLI commands of the `keyshare` and `pep` modules are missing a long message to describe their usage, which could be helpful for users and developers.

Recommendation

We recommend specifying a long message for all transaction and query CLI commands. Each command should provide a description of how to correctly use the command.

Status: Resolved

24. Duplicated `ValidateBasic` invocation in CLI

Severity: Informational

All transaction CLI commands registered in the `GetTxCmd` function in `x/keyshare/client/cli/tx.go` and `x/pep/client/cli/tx.go` call the `msg.ValidateBasic` function before calling `GenerateOrBroadcastTxCLI`.

As `msg.ValidateBasic` is already called inside the `GenerateOrBroadcastTxCLI`, this is an unnecessary and duplicated invocation.

Recommendation

We recommend removing the duplicated `msg.ValidateBasic` invocation.

Status: Resolved

25. Non-standard management of external functions

Severity: Informational

According to the official Cosmos SDK documentation in https://docs.cosmos.network/main/building-modules/keeper, external keepers are listed in the internal keeper's type definition as interfaces. In the audited codebase, however, the keeper's type definition of both the `keyshare` and `pep` modules directly depend on external keepers. This goes against loose-coupling best practices.

Recommendation

We recommend defining the interfaces of external keepers in `expected_keepers.go` and using those.

Status: Resolved

26. Miscellaneous code quality comments

Severity: Informational

Throughout the codebase, instances of unused imports and code have been found.

Recommendation

The following are some recommendations to improve the overall code quality, efficiency and maintainability:

- Remove unused imports in:
  - `x/keyshare/client/cli/query.go:3-14`
  - `x/keyshare/client/cli/tx.go:10`
  - `x/pep/client/cli/querygo:3-14`
- Remove unused code in:
  - `x/keyshare/client/cli/tx_register_validator.go:14`
  - `x/keyshare/client/cli/tx_send_keyshare.go:15`
  - `x/pep/module.go:52-54`
- Remove unused errors in:
  - `x/keyshare/types/errors.go:11` and `14`
  - `x/pep/types/errors.go:11-12`
- Remove unused functions in:
  - `x/keyshare/types/expected_keepers.go:12` and `18`
  - `x/pep/types/expected_keepers.go:16`
- Remove unused event type in `x/pep/types/events_ibc.go:5`.
- Remove `paramstore` of `Keeper` in `x/pep/keeper/keeper.go:22` since there is no global module param.

Status: Resolved

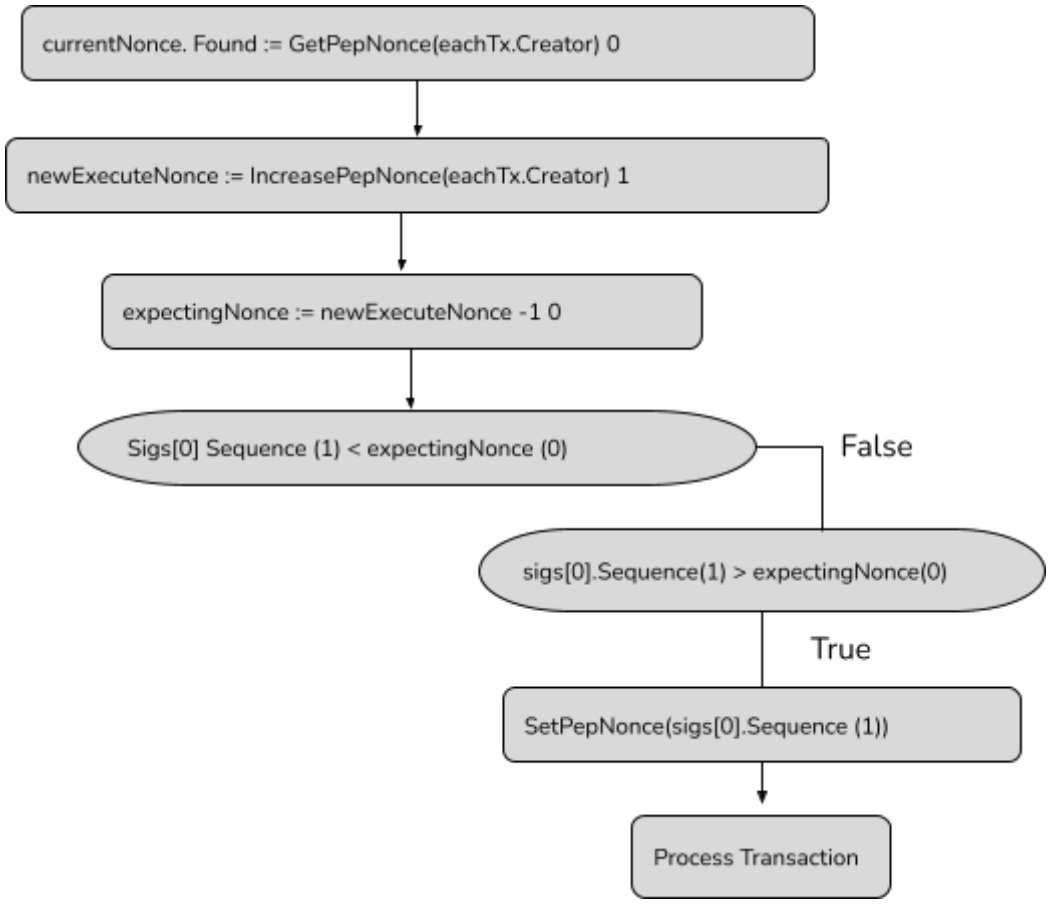
Appendix: Test Cases

1. **Test case for "Insufficient validation of `PepNonce` can lead to transaction replay attack"**

Below is the dry run of `PepNonce` replay. User signs the first transaction with sequence 1 meaning `sigs[0].sequence` is set to 1.

The numerical values at the end of boxes are the result of dry run. we can see that same transaction is replayed in diagram 2 since `if(sigs[0].sequence < expectingNonce)` can be bypassed.

- Transaction executed 1st time:



- Transaction replayed: