Security Audit Report

Interblockchain Communication Protocol: Specification and Code

January 19, 2021

Contents

Audit overview . . . 4
Audit Dashboard . . . 7
Engagement Goals . . . 9
Coverage . . . 10
Recommendations . . . 11
- Short term . . . 11
- Long term . . . 12

Findings . . . 14
Specification Findings . . . 16
- ICS20 - Refund logic differs between code and specification . . . 16
- Faulty semantics underspecified . . . 18
- ICS04 - Incorrect Properties . . . 20
- ICS18 - Relayer underspecified . . . 22
- ICS02 - Suggestions for restructuring . . . 24
- ICS20 - Specification allows token lost issue in the crossing hellos scenario . . . 26
- ICS20 - Type mismatch for amount packet field between code and spec . . . 28

Code Findings . . . 30
- ICS07 - Tendermint Client: wrong usage of unbonding period . . . 30
- Malicious IBC app module can claim any port or channel capability using LookupModuleByPort/ByChannel . . . 32
- ICS20 - Escrow address collisions . . . 36
- ICS03/04 - Crossing hellos with fixed identifier are not live . . . 41
- Non-atomicity of operations . . . 44
- ICS20 - Model and Model-based Tests for Token Transfer . . . 48
- ICS20 - Panic on receiving multi-chain denominations . . . 51

Audit overview

In December 2019, the Interchain Foundation engaged Informal Systems to take leadership over the implementation of the Inter-Blockchain Communication (IBC) Protocol in Rust and to formally specify the protocol in TLA+. Starting October 26 2020, the Informal Systems team conducted an internal audit of the existing IBC specification in English and implementation in Go. The audit was conducted over the course of eight person-weeks with four research engineers. Note that engineers involved in the audit were primarily those that had not worked on the Rust and TLA+ IBC deliverables; that said, some members of the audit team already had a good understanding of the IBC protocols and familiarity with the SDK code base.

We audited the relevant components in the IBC directory of the Cosmos-SDK, working from commit hash 6cbbe0d, and the corresponding IBC specification, working from commit hash 7e6978a. The `github.com/cosmos/relayer` repository, which implements the IBC relayer in Golang, was not in the scope. Throughout the process, we worked closely with the Interchain GmbH team in order to continuously integrate the outputs of the audit, so the code and the specification were moving targets during the audit. We want also to note that many of our recommendations were already addressed in the meantime.

The audit was conducted in a top down approach, starting from the implementation of the token transfer application (ICS20), and moving down the IBC stack analysing the implementations of channels and packets (ICS04), connections (ICS03) and clients (ICS02 and ICS07). For each ICS we started by thoroughly inspecting the specification; we formalized the protocol by writing pre-conditions and post-conditions of the core functions that constitute the protocol (see models directory). We then reviewed the code with a focus on finding (a) discrepancies between the code and the specification and, (b) possible vulnerabilities.

The audit of the Token Transfer application (ICS20) surfaced several issues:

- IF-IBC-01, IF-IBC-02, IF-IBC-06 and IF-IBC-07 pointed out the discrepancies between the implementation and the specification that might lead to some attack scenarios due to user misunderstanding the specification. Furthermore, it could lead to the insecure (future) implementations of the protocols.
- IF-IBC-10 captures the protocol/implementation bug related to escrow address collisions
- IF-IBC-12 points to some poor software practices and lack of proper documentations.
- IF-IBC-13 and IF-IBC-14 capture the initial work on model based testing (MBT) of token transfer application, which caught a panic triggered by multi-chain token denominations.

After ICS20, we moved to channels and packets (ICS04). In addition to the imprecise specifications and discrepancies between the implementations and the specification (IF-IBC-02), we realised

that the definitions of properties were imprecise and that they might mislead users, leading to risky scenarios that could result in loss of funds (IF-IBC-03). Furthermore, we identified a protocol/implementation bug whereby a malicious relayer could prevent the connection and channel handshakes from terminating (IF-IBC-11).

In parallel with the audit of the ICS04 implementation, we took a more general look at the object capabilities implementation in the Cosmos SDK as it is a critical security component. We captured some issues in IF-IBC-09 in the context of the IBC port and channel keepers, although it can be seen as a more general issue of the SDK's object capability system.

Apart from IF-IBC-11, no major issues were found with respect to ICS03. With respect to clients (bottom of the IBC stack), we found a protocol bug in the Tendermint light client implementation (IF-IBC-08), and suggested major restructuring of ICS02 in IF-IBC-05 to improve its clarity and rigor. Furthermore, we reviewed ICS18 (though we haven't carefully reviewed Go implementation of the relayer), and wrote up our findings in IF-IBC-04; in short, the relayer logic is significantly underspecified with several important points left open, that could lead to wrong implementations.

Considering the importance and security risks of Token Transfer (ICS-20), we have invested some time into developing model based tests based on TLA+ specifications. The approach taken is captured in IF-IBC-13 and the bug found using MBT is explained in IF-IBC-14.

Note that two additional aspects of IBC were not covered in this audit. The first is ICS23 and its implementation. The second is the upgrade logic for connections and channels, which is not captured in the specification. We recommend the upgrade logic be captured more clearly in the specification, and that both these aspects of the specification and code receive further review.

In addition to those major findings, we have created several issues on both cosmos-sdk and cosmos/ics repositories and engaged with the team at Interchain Berlin in helping them address the most critical ones.

Overall, our team found the protocol to be well designed and implemented. However, we anticipate that early IBC adopters may have a hard time correctly navigating through and understanding the specification, i.e., the specification could benefit from improved clarity and better organisation. Furthermore, misunderstanding of the guaranteed properties might lead to insecure implementations and wrong usage. We have made several concrete recommendations in this report how this can be improved. At the implementation level, the major issues come from the integration of the IBC implementation in the Cosmos SDK framework. This made it hard to understand the execution model in which IBC handlers run, the atomicity assumptions of functions and error handling and propagation. Furthermore, the implementation of capabilities framework might lead to security issues where a malicious IBC module could be able to obtain the capability associated with any channel, and send/receive messages on another modules channel. Although code has pretty good

test coverage, our recommendation is to take advantage of the TLA+ specifications of the IBC protocol to implement model based tests (MBT) for the complex corner cases of the critical components of the stack, following the example demonstrated during the audit. Finally, we want to note that although we have found several discrepancies between the code and the specification, the code had usually implemented things correctly or took other defensive measures.

Audit Dashboard

Target Summary

- **Name**: Interblockchain Communication Protocol
- **Code Version**: 6cbbe0d4ef90f886dfc356979b89979ddfcd00d8
- **Specification Version**: 7e6978ae551bbed439c69178184dea0a25d0e747
- **Type**: Implementation
- **Platform**: Golang

Engagement Summary

- **Dates**: October 26 through November 19 2020
- **Method**: Whitebox
- **Employees Engaged**: 4
- **Time Spent**: 8 person-weeks

Vulnerability Summary

Issue Type	#	Finding
Total High-Severity Issues	4	IF-IBC-06, IF-IBC-09, IF-IBC-10, IF-IBC-14
Total Medium-Severity Issues	5	IF-IBC-01, IF-IBC-02, IF-IBC-03, IF-IBC-08, IF-IBC-11
Total Low-Severity Issues	2	IF-IBC-04, IF-IBC-07
Total Informative-Severity Issues	3	IF-IBC-05, IF-IBC-12, IF-IBC-13
Total	**14**	

Category Breakdown

Finding Type	#
Specification deviation	6
Protocol/Implementation bug	3
Implementation bug	2
Specification restructuring proposals	1
Code restructuring proposals	2
Total	**14**

Severity Categories

Severity	Description
Informational	The issue does not pose an immediate risk (it is subjective in nature); they are typically suggestions around best practices or readability
Low	The issue is objective in nature, but the security risk is relatively small or does not represent security vulnerability
Medium	The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited
High	The issue is exploitable security vulnerability

Engagement Goals

This audit was scoped by the Informal Systems team in order to assess the correctness and security of the IBC Go implementation. The timing of this internal audit coincides with the upcoming deployment of the IBC protocol to the Cosmos Hub, which marks a critical release in the evolution of the Cosmos Network – the ability for arbitrary blockchains to communicate with one another. The primary focus of the audit was on the upcoming Stargate launch, but we also reviewed the code and specification from the IBC as a development environment perspective for cross-chain applications. Therefore, not all findings present issues with the existing code but might become security vulnerability in the context of IBC applications and new implementations of the IBC specifications.

Specifically, during the audit, we sought to answer the following questions:

- Is the protocol defined unambiguously?
- Are there discrepancies between the code and the specifications?
- Are the stated properties and invariants ambiguous? Can we find violations of the properties and invariants?
- Are IBC messages correctly validated?
- How is error handling and checking state validity done? Are invalid transactions handled correctly? Can malicious inputs cause crashes or invalid states?
- Is the code/specification organized in a way that simplifies reviews?

Coverage

Informal Systems manually reviewed the relevant components in the IBC directory of the Cosmos-SDK starting at commit hash 6cbbe0d. Manual review resulted in findings IF-IBC-001 through IF-IBC-012. We also did a preliminary model-based testing of the ICS-20 token transfer app based on the TLA+ specification of the ICS20 that resulted in IF-IBC-13 and IF-IBC-14. This audit focused on implementation and specification of token transfer application (ICS20), channels and packets (ICS04), connections (ICS03), clients (ICS02 and ICS07), and the relayer specification (ICS18), but the relayer implementation was not in the scope.

A non-exhaustive list of some approaches taken, and their results include:

- Capturing pre and post conditions of the core IBC handlers helped us identify ambiguities and bugs in the IBC specifications that could lead to invalid usage of IBC and insecure IBC implementations (IF-IBC-01)
- Carefully reviewing (manually) the code and the specifications led to IF-IBC-06 and IF-IBC-07.
- Understanding execution model and implementation of object capabilities in SDK helped us identify the following issues: IF-IBC-02, IF-IBC-09, IF-IBC-10 and IF-IBC-12.
- Analysing handler executions under the worst case scenarios (allowed by the model) confirmed that, besides IF-IBC-08 and IF-IBC-11, which were addressed in the meantime, protocols are safe under the threat model assumed. Note that ensuring protocol liveness heavily depends on the correct relayer implementations, and it was left off the scope of this audit.
- Using model based testing (IF-IBC-13), where tests are automatically generated using Apalache model checker from TLA+ specification, surfaced IF-IBC-14, and suggests that it might be useful to expand this exercise further in the future.
- Finally, carefully reading specifications in order to understand expected behaviour led to several findings that suggest possible improvements in making properties more clear (IF-IBC-03), relayer specification more complete (IF-IBC-04) and restructuring of ICS02 in order to simplify onboarding of IBC adopters (IF-IBC-05).

Recommendations

This section aggregates all the recommendations made during the audit. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

- **Separate application-level acknowledgement codes from low level infrastructure aborts** (IF-IBC-01). The easiest way to fix this issue would be aligning specification of ICS20 with the code.
- **Distinguish between valid and invalid counterparty semantics in all function definitions of ICS20** (IF-IBC-02). A user not aware of the Byzantine semantics of `recvPacket` may not be aware of this, which hinders a proper risk assessment, and development of application-level counter measures. Not taking this into account opens an area of attack that may lead to substantial financial loss. Furthermore, in the discussion be precise about what is meant when referring to actions on the counterparty. E.g., make clear what is meant by "sent" in "IBC packet sent on the corresponding channel end on the counterparty chain".
- **Provide more precise properties in ICS04** (IF-IBC-03). Application developer may be lured into the trap of assuming these wrong properties and building their application on top of it, which opens a wide range of exploitable attack scenarios. Also distinguish between properties between two valid chains, and properties a valid chain can expect if the counterparty chain is invalid (Byzantine).
- **Make explicit all the ordering constraints** (IF-IBC-04). The order in which datagrams are submitted is crucial to ensure progress in IBC. An exhaustive representations of these constraints need to be made explicit in the relayer specification (ICS018). Otherwise, it might lead to the incorrect relayer implementations that fail to ensure liveness, or results in transaction fees being spent unnecessarily. Furthermore, the required behavior of the relayer for timeout handling should be specified.
- **Improve ICS020 specification to prevent token lost issue in the crossing hellos scenario** (IF-IBC-06). In `onChanOpenTry`, the escrow account should be created only if it does not exist, i.e., a check should be added to create an escrow account only if `channelEscrowAddresses[channelIdentifier]` does not exist.

- **Align the type of `FungibleTokenPacketData.Amount` in ICS020 specification and implementation** (IF-IBC-07)
- **Correct wrong usage of unbonding period in the Tendermint client** (IF-IBC-08). Document and specify the misbehavior treatment, and make explicit timing assumptions. Change the code to

  ```
  if currentTimestamp.Sub(consState.Timestamp) >= clientState.TrustingPeriod {
  ```

  instead of

  ```
  if currentTimestamp.Sub(consState.Timestamp) >= clientState.UnbondingPeriod {
  ```

- **Correct handshake liveness issue with crossing hellos in ICS03/ICS04** (IF-IBC-11) Add a mechanism in both the specification, and the implementation to deal with mismatched parameters.
- **Document in the developer documentation the implicit assumption on error propagation up the stack** (IF-IBC-12). At the moment, there are only hints regarding this in the form similar to this paragraph in the OnRecvPacket. Such hints do not constitute enough developer guidance to avoid introducing severe bugs, especially for Cosmos SDK newcomers.

Long term

- **Improve ICS 20 implementation to serve as a template for future IBC applications** (IF-IBC-01). As ICS 20 also will serve as a template for future IBC applications, a clearer separation between application-level errors and infrastructure roll-backs (and panics) would be advantageous. For that purpose we suggest a more robust implementation of token transfer that does not rely on the bank module panicking.
- **Provide test cases that involve corner cases** (IF-IBC-04). What complicates the situation with testing relayer is the fact that ordering constraints involve concurrency effects that should be mitigated (serialized) at the relayer. Such issues are typically hard to reproduce or debug. Relying on model-based testing (IF-IBC-13) could be useful in this context.
- **Major refactoring of ICS02 specification** (IF-IBC-05). ICS02 (due to its number) serves as de facto entry point for newcomers who want to learn about IBC. The current structure of the text does not serve that purpose well. We suggest a major reorganization, perhaps along the following ideas in IF-IBC-05.

- **Prevent malicious IBC app module to claim any port or channel capability** (IF-IBC-09) Using `LookupModuleByPort/ByChannelKeeper` methods should be restricted from outside the module - whoever is composing modules, presumably in `app.go`, should explicitly define which methods of a keeper each module gets. Otherwise, a malicious IBC application module could add the `LookupModuleByPort` method to its expected `PortKeeper` interface, and then open channels on some other module's port. This would allow a malicious IBC module to grab the capability associated with any channel, and send/recv messages on another module's channel.
- **Prevent escrow address collisions in ICS20** (IF-IBC-10). In order to mitigate the address collision problems, several recommendations are made:

  (1) limit the pre-image space,
  (2) use slow hash function,
  (3) change the channel establishment protocols and/or
  (4) redesign the Cosmos address space.

- **Document non-atomicity assumptions of operations** (IF-IBC-12). Either make the SDK functions atomic or introduce a separate explicit step for handlers, say `CommitState`, that the handler will need to call to write state changes to the store. Otherwise, non-atomicity of operations can lead to bugs.
- **Use model based testing on critical components, ICS20 and ICS04** (IF-IBC-13). We have demonstrated on the ICS20 example the benefits of model-based testing. The set of developed artifacts covers the efforts of creating a preliminary model and a set of model-based tests for ICS-20 token transfer. While our model-based tests for the token transfer module have been successfully merged into Cosmos-SDK (see the PR Cosmos-SDK #8145), only a very limited number of tests are contained there. We recommend the Cosmos-SDK developers to master the model-based testing methodology (see IF-IBC-13 for a short description), and to extend their test suite with more model-based tests. This should help both to reduce the testing efforts from the developers, and to increase the coverage of complicated scenarios.

Findings

#	Title	Type	Severity	Code or Spec
IF-IBC-01	Refund logic differs between code and specification	Specification deviation	Medium	Spec
IF-IBC-02	Faulty semantics underspecified	Unclear Specification	Medium	Spec
IF-IBC-03	Incorrect Properties	Specification error	High	Spec
IF-IBC-04	Relayer underspecified	Unclear specification	Low	Spec
IF-IBC-05	Suggestions for ICS 02 Restructuring	Specification restructuring proposal	Informative	Spec
IF-IBC-06	Token lost issue in the crossing hellos scenario	Specification error	High	Spec
IF-IBC-07	Type mismatch	Specification deviation	Low	Spec
IF-IBC-08	Wrong usage of unbonding period	Protocol Bug	Medium	Code
IF-IBC-09	Malicious IBC app module can claim any port or channel capability	Implementation Bug	High	Code
IF-IBC-10	Escrow address collisions	Protocol / Implementation Bug	High	Code
IF-IBC-11	Crossing hellos with fixed identifier are not live	Protocol / Implementation Bug	Medium	Code
IF-IBC-12	Non-atomicity of operations	Bad Software Practice / Lack of Documentation	Informative	Code

#	Title	Type	Severity	Code or Spec
IF-IBC-13	Model-based Testing for Token Transfer	Set of new artifacts to facilitate rigorous testing	Informative	Code
IF-IBC-14	Panic on receiving multi-chain denominations	Implementation Bug	High	Code

Specification Findings

IF-IBC-01

ICS20 - Refund logic differs between code and specification

Severity: Medium
Type: Specification deviation
Difficulty: Unclear
Involved artifacts: ICS 20, applications/transfer/module.go

Description

In the specification ICS20 `recvPacket` in the case where there is an error within the bank module (`TransferCoins` and `Mintcoins`), an acknowledgement with error code is generated. As a result, at the sending chain the function `onAcknowledgePacket` is executed with a "non-success" code (assuming "normal" operation with a reliable relayer), which in turn calls `refund`.

In the implementation, in case of an error, the bank module panics, the transaction is aborted. In case of no other events (e.g., later a packet removes money from some account which results in the original packet going through upon retry without panic), on the sender chain, the function `onTimeoutPacket` is executed which in turn performs `refund`.

Problem Scenarios

It seems that the intuition of the logic in the specification is to highlight that an acknowledgement value can be used to control the application. For instance, instead of refunding, one might think of retrying sending.

From a design viewpoint, separating application-level acknowledgement codes from more low level aborts seems advantageous. However, it seems as if the bank module does not provide this fine-grained distinction.

Recommendation

The easiest fix would be to - Change specification to align with code. - Due to the discrepancy it appears that no acknowledgement with a code different from "result" is currently sent. If this was true, the error branch in OnAcknowledgementPacket would be unreachable code. We suggest to check that, and if this is the case, it should be removed from the specification and the code.

However, as ICS 20 also will serve as a template for future IBC applications, a clearer separation between application-level errors and infrastructure roll-backs (and panics) would be advantageous. For that purpose we suggest a more robust implementation of token transfer that does not rely on the bank panicking.

IF-IBC-02

Faulty semantics underspecified

Severity: Medium
Type: Unclear Specification
Difficulty: Unclear
Involved artifacts: ICS 02, ICS 03, ICS 04, ICS 20

Description

Throughout the documents, the environment in which the functions are called are often unclear and sometimes misleading. For instance, in ICS 04 the specification of the function `sendPacket` is followed by the description of receiving packets. It starts with

> The `recvPacket` function is called by a module in order to receive an IBC packet sent on the corresponding channel end on the counterparty chain.

In the context of the specification, an application developer is led to believe that "sent on the corresponding channel" means that `sendPacket` has been executed at the counterpart chain.

However, IBC is designed also to provide some services if the counterparty chain performs invalid transitions. This should be made explicit so that an application developer is not lured into the trap of assuming a friendly environment. It should be distinguished between semantics to expect from a valid counterparty and semantics to expect (or not expect) from an invalid (Byzantine) counterparty.

Problem Scenarios

In ICS 20, the fungible token transfer, if a receiver receives a token with prefixed denomination, the validity and fungibility of a token depends on the validity of all the chains encoded in the prefix. In particular, an invalid chain B can send an arbitrary number of prefixed chain A tokens to chain C out of thin air, while none of these tokens are escrowed at A. A user not aware of the Byzantine semantics of `recvPacket` may not be aware of this, which hinders a proper risk assessment, and development of application-level counter measures. Not taking this into account opens an area of attack that may lead to substantial financial loss.

Recommendation

- In all function definitions, distinguish between valid and invalid counterparty semantics.
- In the discussion be precise about what is meant when referring to actions on the counterparty. E.g., make clear what is meant by "sent" in "IBC packet sent on the corresponding channel end on the counterparty chain".

IF-IBC-03

ICS04 - Incorrect Properties

Severity: Medium
Type: Specification error
Difficulty: Hard
Involved artifacts: ICS04

Description

ICS 04 provides a list of 6 "desired properties". The first four of which are misleading or underspecified:

> - The speed of packet transmission and confirmation should be limited only by the speed of the underlying chains. Proofs should be batchable where possible. Exactly-once delivery

- "Speed" is too vague.
- The formulation ignores relayers that are the active components in packet transmission.

> - IBC packets sent on one end of a channel should be delivered exactly once to the other end.

This property are also vague as:

- in the absence of a relayer no packets can be delivered
- ignores timeouts
- unspecific what "sent" means, cf. IF-IBC-02.

> - No network synchrony assumptions should be required for exactly-once safety. If one or both of the chains halt, packets may be delivered no more than once, and once the chains resume packets should be able to flow again.

- It is unclear what "exactly-once safety" is.
- Network synchrony assumptions (e.g. trusting periods) may be necessary for client safety (ICS 02 and ICS 07).
- If both chains halt for too long, light clients may not accept headers, and manual (subjective) initialization is needed.

- On ordered channels, packets should be sent and received in the same order: if packet x is sent before packet y by a channel end on chain A, packet x must be received before packet y by the corresponding channel end on chain B.

It is not clear what "if packet x is sent before packet y by a channel end on chain A" meant in a context where chain A performs invalid transitions: then a packet with sequence number *i* can be sent after *i+1*. If this happens, the IBC implementation may be broken (depends on the relayer).

Missing properties. "Safety of communication" is not specified, that is, if the counterparty is valid (and there is not attack), the package received was sent using `SendPacket`.

Problem Scenarios

Application developer may be lured into the trap of assuming these wrong properties and building their application on top of it, which opens a wide range of exploitable attack scenarios, for instance:

- The missing safety property might result in accepting a packet without understanding the risk, and thus substantial financial loss. E.g., if it is a fungible token transfer packet (ICS 20), and the counterparty is Byzantine, I might accept a forged token that is not escrowed anywhere.
- Implying that in all cases of two chains stop operation may continue automatically, may lead to not making the application code ready for social recovery.

Recommendation

Provide more precise properties. Also distinguish between properties between two valid chains, and properties a valid chain can expect if the counterparty chain is invalid (Byzantine).

IF-IBC-04

ICS18 - Relayer underspecified

Severity: Low
Type: Unclear specification
Difficulty: medium
Involved artifacts: ICS 18

Description

The relayer is the active component in IBC. It is responsible to submit datagrams in time, and in observing the state of the chains it connects. There are some points that need clarification

> There are implicit ordering constraints.

The order in which datagrams are submitted is crucial to ensure progress in IBC. An exhaustive representations of these constraints need to be made explicit.

That these constraints are not explicitly provided leads to misleading statements in the same document, as the following:

> "Race conditions": "if two relayers do so, the first transaction will succeed and the second will fail."

This statement is only true if both relayers adhere to these "implicit" ordering constraint that the header need to be installed first. It might be that the first relayer fails, then the header is installed, then the second relayer succeeds.

In addition, timeout handling is not discussed.

Problem Scenarios

- If the relayer has to create datagrams for two packets on an ordered channel with sequence number 4 and 5, and submits them in wrong order (5 before 4), the check in the packet handler will fail on both packets.
- The above scenarios considers just one ICS (ICS 04), while the example given in ICS 18 (header before packet) highlights an ordering between different ICSs. They are especially hard to infer.

Having these order constraints underspecified might lead to relayer implementations that:

- from the protocol viewpoint do not ensure liveness in communication,
- from the economic viewpoint results in transaction fees being spent unnecessarily.

Recommendation

What complicates the situation is that these ordering constraints involve concurrency effects that should be mitigated (serialized) at the relayer. Such issues are typically hard to reproduce or debug.

- Make explicit all the ordering constraints.
- Provide test cases that involve corner cases.
- Specify the required behavior of the Relayer for timeout handling.

IF-IBC-05

ICS02 - Suggestions for restructuring

Severity: Informative
Type: Specification restructuring proposal
Difficulty: hard
Involved artifacts: ICS 02

Description

ICS002 (due to its number) serves as de facto entry point for newcomers who want to learn about IBC. The current structure of the text does not serve that purpose well. We suggest a reorganization, perhaps along the following ideas (if such a document already exists it should just be linked in ICS 002).

Problem Scenarios

N/A

Recommendation

New entry point Create a new entry point into IBC specification. In this new document:

- start with an example:
  - two applications A1 and A2 on two chains C1 and C2.
  - A1 sends a packet to A2.
  - What does the relayer have to do (reading state, submit packet, submit header, timeouts),
  - how the packet reaches A2, how A2 verifies the packet via channel, connection, client, header.
  - explain that header of proof height must be there in order to verify.
- This new document should also contain a discussion of the blockchain. The current one is unclear. My understanding is that the rest of the ICSs (except the relayer) deal with sequential code, while here necessarily we need to talk about distributed aspects, faults, etc.

So the language of pseudo code hits its limits. As a result, because distributed aspects are not described, it is not clear how one aspect can have a "MUST NOT" and q "Possible violation scenario" at the same time.

- This would also be a good point to clarify the situation of counterparty chains.
  - What are the precise assumptions?
  - What is the good "normal" case?
  - What is if the counterpart chain violates validity?
  - What is a light client attack?

Suggestions for ICS 02 (the client specification)

- Make explicit what function of a client face towards
  - the relayer
  - the connection
  - other?
- More structure: right now "validity predicate", "Misbehaviour predicate", "Height", "ClientState", etc., all appear on the same level, and thus indicate the same importance and same quality. That doesn't help the reader. Add structure by e.g., sections highlighting
  - verification facing towards local channels, connections
  - functions for relayer to submit data (validity predicate, misbehavior predicate)
  - functions for relayer to query information about the client
- Understand who is the expected audience (developers of IBC, developers of relayers, other?). What information are they looking for, how can we make it easy to find it.
- it is a long document. An outline (like that one) would be great.

IF-IBC-06

ICS20 - Specification allows token lost issue in the crossing hellos scenario

Severity: High
Type: Specification error
Difficulty: easy
Involved artifacts: ICS 20, applications/transfer/module.go

Description

The ICS20 specification uses undefined `newAddress()` in onChanOpenInit and `onChanOpenTry` to create an escrow address and stores it in a map under the `channel id` as a key. In the case of crossing-hello scenario, different escrow accounts are created both in `onChanOpenInit` and `onChanOpenTry`, with latter overwriting the former. This can lead to the token lost issue if `createOutgoingPacket` is called in between.

Problem Scenarios

Imagine the following scenario:

- on chain A `onChanOpenInit` is called which leads to the creation of the escrow address `E1` that is stored in `channelEscrowAddresses` under `channelIdentifier` as a key
- on chain A, `createOutgoingPacket` is called, and it moves tokens to the escrow address `E1`
- on chain A, `onChanOpenTry` is called that creates a new escrow address `E2` that replaces `E1` in the `channelEscrowAddresses` map under `channelIdentifier` as a key.
- on chain A, `onRecvPacket` is called to withdraw tokens that are escrowed in `E1` with `createOutgoingPacket`. It will fail as there are no tokens in the escrow account `channelEscrowAddresse` as it points to `E2`, while tokens are in `E1`.

Recommendation

In `onChanOpenTry`, the escrow account should be created only if it does not exist, i.e., a check should be added to create an escrow account only if `channelEscrowAddresses[channelIdentifier]` does not exist.

Note that the in the SDK implementation of the ICS20, code uses a deterministic function to create an escrow account that receives two parameters `portId` and `channelId`. Therefore, this implementation does not suffer from the problem mentioned here as the implementation differ from the specification, i.e., only one escrow account is created in the mentioned scenario.

IF-IBC-07

ICS20 - Type mismatch for amount packet field between code and spec

Severity: Low
Type: Specification deviation
Difficulty: easy
Involved artifacts: ICS 20

Description

The ICS20 specification uses uint256 for FungibleTokenPacketData.Amount

```
interface FungibleTokenPacketData {
  denomination: string
  amount: uint256
  sender: string
  receiver: string
}
```

The code uses uint64

```
type FungibleTokenPacketData struct {
    // the token denomination to be transferred
    Denom string
    // the token amount to be transferred
    Amount uint64
    // the sender address
    Sender string
    // the recipient address on the destination chain
    Receiver string
}
```

The confusion is maybe due the coin type where the amount is a BigInt.

Problem Scenarios

The user might send an amount that is a bigger integer than accepted and have his/her transaction abort due to an overflow.

Recommendation

Change the spec.

Code Findings

IF-IBC-08

ICS07 - Tendermint Client: wrong usage of unbonding period

Severity: Medium
Type: Protocol bug
Difficulty: easy
Involved artifacts: ICS 07, `light-clients/07-tendermint/types/misbehaviour_handle.go`

Description

In the code we observe the following line:

```
if currentTimestamp.Sub(consState.Timestamp) >= clientState.UnbondingPeriod {
```

It should ensure that the consensus state that is used to verify one of the headers that constitute misbehavior is not too old.

According to the Tendermint Security model, validators in NextValidators of a block (header) with time t need to behave correctly until t + *TrustingPeriod*. After that time, they may behave arbitrarily (given that they do not appear in as NextValidator in a later block.). However here we are checking against the UnbondingPeriod, not the TrustingPeriod, where `UnbondingPeriod > TrustingPeriod`. As a result, this check allows nodes that are outside the fault assumption to shut down the client.

Remark. The implemented misbehavior treatment in the Tendermint Client is not specified in ICS 07. We categorize it as "protocol bug" under the assumption that the code line is the result of a protocol design process that was documented somewhere else that led to this conclusion, rather than a mistake in the implementation.

Problem Scenarios

If the age of the `consState` is between *TrustingPeriod* and *UnbondingPeriod* the header will be accepted as base to verify one of the conflicting headers that constitutes misbehavior.

During this period, the validators in `consState.NextValidators` are not considered trustworthy anymore. As we must assume that they behave arbitrary, they can forge the header that is part of misbehavior (there is no incentive not to do that). As a result adversarial former validators may shut down the client without risking anything.

Recommendation

- Document and specify the misbehavior treatment, and make explicit timing assumptions.
- Change the code to

```
if currentTimestamp.Sub(consState.Timestamp) >= clientState.TrustingPeriod {
```

IF-IBC-09

Malicious IBC app module can claim any port or channel capability using LookupModuleByPort/ByChannel

Severity: High
Type: Implementation bug
Difficulty: High
Involved artifacts: app.go, /x/ibc/core/05-port/keeper/keeper.go, x/ibc/core/04-channel/keeper/keeper.go

Description

This is an instance of a more general class of issue with the SDK's object capability system, though it was here surfaced in the context of the IBC Port and Channel Keepers.

In an ocap system, each ocap has associated with it a set of actions that can be performed, ie messages that can be sent or methods that can be called. In the SDK, actions are defined through Keepers. Since all methods on a Keeper are public, any module with access to a keeper can call all methods exposed by that keeper. In each module, we define a `types/expected_keepers.go` which define the keepers this module uses (besides its own) as interfaces with a limited set of methods. While useful for testing and local representation of the actions this module needs to execute, this does not meaningfully restrict the ability of a malicious module to add other methods to its expected keepers interfaces, and thus access other methods exposed by those keepers beyond those which are intended!

Of course this depends on malicious modules including code that escapes audit and review, which is maybe not an overwhelming concern right now, but the SDK has at least intended to mitigate such issues following principles of the least privilege via the ocap model. The problem is that keepers are actually granted to modules without minimizing privilege at all.

For instance, modules which receive the PortKeeper need only the PortKeeper.BindPort method:

```
// PortKeeper defines the expected IBC port keeper
type PortKeeper interface {
    BindPort(ctx sdk.Context, portID string) *capabilitytypes.Capability
}
```

But a full `05-port/keeper/Keeper` is actually passed in, and the full keeper has a `LookupModuleByPort` method which returns the capability associated with a port. If this were exposed to IBC application

modules, they'd all be able to claim capabilities for any other modules port, which would defeat the purpose of the port system.

Problem Scenarios

A malicious IBC application module could add the `LookupModuleByPort` method to its expected PortKeeper interface, and then open channels on some other module's port. For instance the following diff to the `transfer` module should do it:

```
-// BindPort defines a wrapper function for the ort Keeper's function in
+// BindPort defines a wrapper function for the port Keeper's function in
 // order to expose it to module's InitGenesis function
 func (k Keeper) BindPort(ctx sdk.Context, portID string) error {
-       cap := k.portKeeper.BindPort(ctx, portID)
-       return k.ClaimCapability(ctx, cap, host.PortPath(portID))
+       someoneElsesPort := "transfer"
+       _, cap, _ := k.portKeeper.LookupModuleByPort(ctx, someoneElsesPort)
+       return k.ClaimCapability(ctx, cap, host.PortPath(someoneElsesPort))
 }

 // GetPort returns the portID for the transfer module. Used in ExportGenesis
diff --git a/x/ibc/applications/transfer/types/expected_keepers.go \
          b/x/ibc/applications/transfer/types/expected_keepers.go
index 284463350..5350c66bd 100644
--- a/x/ibc/applications/transfer/types/expected_keepers.go
+++ b/x/ibc/applications/transfer/types/expected_keepers.go
@@ -45,4 +45,5 @@ type ConnectionKeeper interface {
 // PortKeeper defines the expected IBC port keeper
 type PortKeeper interface {
        BindPort(ctx sdk.Context, portID string) *capabilitytypes.Capability
+       LookupModuleByPort(ctx sdk.Context, portID string)
+          (string, *capabilitytypes.Capability, error)
 }
```

The same holds true for the `04-channel/keeper/Keeper` which exposes a `LookupModuleByChannel` method. Any IBC module could thus grab the capability associated with any channel, and send/recv on another modules channel.

Recommendation

Keeper methods should be restricted from outside the module - whoever is composing modules, presumably in app.go, should explicitly define which methods of a keeper each module gets. Note this implies that expected keeper interfaces may end up duplicated (ie. once in the actual application for security and once in the module for local clarity/convenience), or may come to live in a trusted alternative place outside the control of an individual module. In any case, we may consider an external Secure interface for the expected keeper (from outside the module), and an insecure Local one (inside the module). So long as keeper variables inhabit a variable of the Secure type before being passed into modules, that should be sufficient. Ie.:

```
/*
    Some keeper defined elsewhere with two methods.
    We only want modules to access the Hi method
*/

type A struct{}

func (A) Hi()  {}
func (A) Bye() {}

/*
    Inside the Module
*/

// Malicious module tries to access the Bye method
type Local interface {
    Hi()
    Bye()
}

// Local function within the module
func LocalA(a Local) {}

/*
    Outside the Module
*/
```

```
// We only want modules to get the Hi method
type Secure interface {
    Hi()
}

func TestHelloWorld(t *testing.T) {

    // if we don't restrict a, the module can access all its methods
    a := A{}
    LocalA(a)

    // if we restrict a to the Secure interface,
    // the module can't access other methods
    // and this fails to compile :)
    var a Secure = A{}
    LocalA(a)
}
```

Related

- Cosmos-SDK #5931 points out an issue around module accounts being access by other modules. Even if module accounts were gated by ocaps, as that issue proposes, a general method which allowed those ocaps to be looked up (like we have in channel and port keepers) would still lead to compromise.
- Cosmos-SDK #7093 points out various issues specific to the BankKeeper (ie. maintaining supply invariants), but also mentions that Keeper methods are not restricted before being passed into modules. So this issue is in some sense a duplicate of that issue, but specific for issues with Port and Channel Keeper.

IF-IBC-10

ICS20 - Escrow address collisions

Severity: High
Type: Protocol/Implementation bug
Difficulty: Medium
Involved artifacts: applications/transfer/types/keys.go, applications/transfer/keeper/relay.go
Issue: Cosmos-SDK #7737

Description

GetEscrowAddress() is the truncated to 20 bytes `SHA256(portID + channelID)`. There are three problems with this approach, which are outlined below, and discussed in depth in the Cosmos-SDK issue #7737.

No domain separation between ports and channels Using string concatenation doesn't separate the domains for ports and channels. Thus, e.g. port/channel combinations `("transfer", "channel")` and `("trans", "ferchannel")` will give the same escrow address, the truncated `SHA256("transferchannel")`. This opens the road to exploits. An exploit is possible if some module with the bank capability is able to choose both the port and the channel ids.

The problem is outlined in the Cosmos-SDK #7737 comment.

Collisions between module account addresses Escrow account addresses have arbitrary alphanumeric strings as pre-images of `SHA-256`, and the post-image size of 160 bits. This combination of the small post-image size and the fast hash function makes the Birthday attack feasible, as the security is reduced to only 80 bits. This means that a collision between two different escrow account addresses can be found after approximately 2^80 guesses. A detailed cost analysis to attack truncated SHA256 was already performed in 2018 in the context of Tendermint, and demonstrates that the attack is also feasible and rather cheap from the cost perspective; this analysis is no less relevant today as it was back then.

Finding a collision means that two different escrow accounts map to the same account address. This can lead to the funds being stolen from the escrow account; see the problem scenario below.

The problem of birthday attacks on module accounts is analyzed in Cosmos-SDK #7737 comment.

Collisions between module and non-module account addresses Public account addresses are constructed as the same truncated to 20 bytes `SHA-256` of the Ed25519 public key. While 160 bits of security is enough to prevent collision attacks on specific public addresses, the security against the Birthday attack is again only 80 bits. Here we find ourselves in a half-Birthday mode, when one address is generated by fast `SHA-256`, and another by a relatively slow `Ed25519 PublicKey` computation. While this scenario is much safer, it still opens a number of possible attacks both on escrow and on public accounts; see the particular scenario below.

Problem Scenarios

Collisions between module account addresses Assume that on `ChainA` there is a collision between two escrow addresses for `Addr == AddressHash("transfer", "channel-ab") = AddressHash("transfer", "channel-am")`. The following scenario becomes then possible:

- Sender `Alice` on `ChainA` sends the ICS 20 token transfer of 1000 `atom` over `channel-ab` to receiver `Bob` on `ChainB`.
- As `atom` is a native denomination, in SendTransfer() it gets escrowed and sent to the escrow address `Addr == AddressHash("transfer", "channel-ab")` The ICS 20 packet is sent to `ChainB`
- Mallory discovers the address collision, sets up the fake `ChainM`, creates a connection to `ChainA`, and establishes a channel to the `transfer` module on `ChainA` such that the channel id on `ChainA` is `channel-am`, and on `ChainM` side is `channel-ma`.
- Mallory sends the ICS 20 packet from `Mallory` on `ChainM` to `Mallory` on `ChainA` of 1000 coins with the denomination `transfer/channel-ma/atom`
- OnRecvPacket() recognizes the packet as the funds that originally were transferred from `ChainA` to `ChainM` due to the constructed denomination. It removes the prefix, unescrows 1000 `atom` from `Addr == AddressHash("transfer", "channel-am")`, and happily sends them to `Mallory` on `ChainA`. Unescrowing succeeds because of the address collision and enough funds with the proper denomination on that address.
- Mallory now has 1000 atoms stolen from the original escrow account `Addr == AddressHash("transfer", "channel-ab")`. She demolishes `ChainM`, and liquidates the funds from `Mallory` on `ChainA`. The attack succeeds.

Collisions between module and non-module account addresses Assume that on `ChainA` there is a collision between the escrow address for `("transfer", "channel-ab")` and the ordinary account address obtained from the private key `PrivM`, i.e. `Addr == AddressHash("transfer", "channel-ab") = AddressHash(PublicKey(PrivM))`. The following (trivial) scenario becomes

then possible:

- Sender `Alice` on `ChainA` sends the ICS 20 token transfer of 1000 `atom` over `channel-ab` to receiver `Bob` on `ChainB`.
- As `atom` is a native denomination, in SendTransfer() it gets escrowed and sent to the escrow address `Addr == AddressHash("transfer", "channel-ab")` The ICS 20 packet is sent to `ChainB`
- Mallory discovers the address collision, i.e. it finds the private key `PrivM`.
- Mallory liquidates the funds from `Addr` without any problems, as she posesses the private key for that address. The attack succeeds.

Recommendation

The problem of missing domain separation between ports and channels was already addressed in Cosmos-SDK PR #7960.

We propose four different approaches to mitigate the address collision problems:

Approach 1: limit the pre-image space Do not allow unbounded or very large preimage spaces for module account addresses. Construct module account addresses not from arbitrary strings, but choose parameters from a small set of alternatives. E.g., in case of escrow addresses, the parameters for constructing an escrow address could be:

- a prefix and a version number
- fix PortId to "transfer"
- deterministically construct ChannelId from e.g. a combination of the Ids of communicating chains and a bounded sequence number, e.g. 1-999. E.g. the escrow address could be `AddressHash("ibc-v1/transfer/cosmos-ethereum-001")`

For pre-image spaces up to approximately 32 bits it should be possible to prevent any kind of collisions completely, by enumerating all module account addresses and rejecting any attempt to register another account with the same address.

Approach 2: use slow hash function If limiting the pre-image space is undesirable from the design point of view, then excluding collisions completely becomes infeasible due to high resource consumption (both time and memory), but finding the collision for a dedicated attacker would be still feasible. In that case the alternative recommendation is as follows:

- Change the hashing function from fast SHA-256 to some slow one, e.g. include `Ed25519 PublicKey` computation into the pre-image.
- Perform collision search with limited memory for some limited amount of time – this will provide some probabilistic guarantees of collision absence.

Approach 3: change the channel establishment protocols Implement protocol-level measures to prevent free choice of addresses by a potential attacker. This is more long-term; the goal here is to guarantee the absence of collisions by changing the protocols. One such protocol-level measure could be to implement window-based selection of channel ids. E.g., when establishing a channel each party provides a window of local channel ids, (say [1-1000)), and the counterparty selects one id from that window. In that way the space of channel ids can be left unrestricted, but whenever a channel is established, provide a window that guarantees absence of collisions with any of existing escrow addresses: this will be easy and fast to do, because the windows will be in the order of e.g. bitwidth 10. At the same time, each party can prevent the other one from selecting a specific id, by choosing one from that window that guarantees absence of collisions also to the other party.

Approach 4: redesign the Cosmos address space All of the above problems stem from the fact that a Cosmos address is a single binary piece of data only 20 bytes long. This was already raised before, see e.g. Cosmos-SDK issue #3685. Architecturally cleanest, but also probably most difficult to implement, is to do the following:

- Change the address format to allow multiple address kinds. That way escrow addresses will be a separate address kind, and no overlapping with the public addresses will be possible.
- Extend the address length to 40 bytes, or simply make it (bounded) variable length. While 20 bytes are still enough at the present point, taking into account the speed of the parallelization and miniaturization happening now, moving from 20-byte addresses to at least 32-byte addresses is bound to happen in the next decade.

Concretely, the Cosmos address could look like that (with fields separated by e.g. a 0 byte, and Bech32-encoded):

1. variable-length address differentiator (e.g. public vs escrow).
2. variable-length address data (e.g. `PublicKey` for public addresses, or `PortId/ChannelId` for escrow addresses).

This will bring the following advantages:

- **Remove unneeded complexity**:

  - there will be no need to hash public keys, which are 32 or 33 bytes long, just store them as is.
  - e.g. for escrow addresses, do not hash, also store the port id and channel id directly.
- **Allow unrestricted address flexibility**:
  - different key types – no problem, just register different descriptors, like `secp256k1` or `secp256r1`.
  - group accounts, "organization" type entities, smart contracts, whatever else: request a new descriptor for it, and define the data.
  - one other possibility would be e.g. to have an IPv6 address + 20-byte old-style address to allow so to say "physical" addresses.
- **Greatly increase security**:
  - no collisions between different address types is possible.
  - individual address security is increased to the maximum, e.g. to 32 bytes for Ed25519 addresses.
  - compromising one address type (e.g. the pre-image attack) will have no influence on other types.

It is important that this structure is in the *address post-image*, not in the pre-image. This provides the separation of security between various address kinds.

The detailed discussion on changing the Cosmos address format can be found in the Cosmos-SDK issue #5694

Extended analysis for Approach 1: limit the pre-image space While preparing this audit report, the Cosmos SDK development team has decided to follow the first of the proposed approaches, and limit the pre-image space for escrow addresses to 32 bits; this has been addressed by the Cosmos SDK PR #7967. From our side, we have performed exhaustive search for escrow address collisions within this pre-image space, and have proven their absence. The details can be found in the Cosmos-SDK #7737 comment.

IF-IBC-11

ICS03/04 - Crossing hellos with fixed identifier are not live

Severity: Medium
Type: protocol/implementation bug
Difficulty: medium
Involved artifacts: ICS 03 specification, ICS 04 specification, ibc/core/03-connection/keeper/handshake.go ibc/core/04-channel/keeper/handshake.go

Description

When two chains want to open a connection (channel), they may both initialize their connection (channel) ends by calling `connOpenInit` (`chanOpenInit`), before a correct relayer creates a `ConnOpenTry` (`ChanOpenTry`) datagram for either chain. In this scenario, in both the code and specification of the connection (channel) handshake protocols, the two chains may initialize their connection (channel) ends with parameters that do not match. This causes both chains to abort when calling `connOpenTry` (`chanOpenTry`). Thus, their already initialized connection (channel) ends remain in state `INIT` forever, which violates the following liveness property:

> If a connection (channel) end is initialized on a chain, then eventually both the connection (channel) end, stored on the chain, and the connection (channel) end, stored at its counterparty, are open.

In the following, we discuss this issue by focusing on the specification of the connection handshake; the discussion for the channel handshake, as well as the implementations, is analogous. We show a scenario where two chains want to open a connection but have mismatched client identifiers. Observe that this issue may arise not only in cases where client identifiers are mismatched, but also when connection, channel, port identifiers, versions, prefixes, or orderings coming from a `ConnOpenTry` (`ChanOpenTry`) datagram do not match the values stored in the existing connection (channel) end on the receiving chain.

Problem Scenarios

When two chains want to open a connection, they may both initialize their connection ends by calling `connOpenInit`, which may result in a chain assigning values to the fields in its connection end that do not match the values of the respective fields in the counterparty connection end

(even if the connection and counterparty connection identifiers match). In this scenario, once a correct relayer creates a `ConnOpenTry` datagram for each chain, the call to `connOpenTry` fails on at least one of the chains. This implies that the connection handshake does not progress, and the above liveness property is violated.

In more detail, consider the scenario where two chains, `"ChainA"` and `"ChainB"`, want to open a connection. Suppose that `"ChainA"` has two clients for `"ChainB"`, identified by the client identifiers `"clientB1"` and `"clientB2"`. Suppose that `"ChainB"` has a single client for `"ChainA"`, identified by the client identifier `"clientA1"`. To open a connection, both chains execute the following steps:

1. `"ChainA"` calls `connOpenInit` and initializes its connection end with the following values:
   - connection identifier: `"connAtoB"`,
   - counterparty connection identifier: `"connBtoA"`,
   - client identifier: `"clientB1"`,
   - counterparty client identifier: `"clientA1"`,
2. `"ChainB"` calls `connOpenInit` and initializes its connection end with the following values:
   - connection identifier: `"connBtoA"`,
   - counterparty connection identifier: `"connAtoB"`,
   - client identifier: `"clientA1"`,
   - counterparty client identifier: `"clientB2"`,
3. a correct relayer creates a `ConnOpenTry` datagram for `"ChainB"` by scanning `"ChainA"`'s state, where the field `clientIdentifier` is set to `"clientA1"`, and the field `counterpartyClientIdenti` is set to `"clientB1"`.
4. a correct relayer creates a `ConnOpenTry` datagram for `"ChainA"` by scanning `"ChainB"`'s state, where the field `clientIdentifier` is set to `"clientB2"`, and the field `counterpartyClientIdenti` is set to `"clientA1"`.
5. `"ChainB"` receives the `ConnOpenTry` datagram and calls `connOpenTry`. A connection end on `"ChainB"` is already initialized, and the connection and counterparty connection identifiers match those from the `ConnOpenTry` datagram. However, the existing connection end has `counterpartyClientIdentifer` set to `"clientB2"`, which does not match the identifer `"clientB1"`, coming from the field `counterpartyClientIdentifer` from the `ConnOpenTry` datagram.
6. `"ChainA"` receives the `ConnOpenTry` datagram and calls `connOpenTry`. A connection end on `"ChainA"` is already initialized, and the connection and counterparty connection identifiers match those from the `ConnOpenTry` datagram. However, the existing connection end has `clientIdentifer` set to `"clientB1"`, which does not match the identifer `"clientB2"`, coming from the field `counterpartyClientIdentifer` from the `ConnOpenTry` datagram.

The call to `connOpenTry` thus fails here, on both sides:

```
abortTransactionUnless(
      (previous === null) ||
      (previous.state === INIT &&
        previous.counterpartyConnectionIdentifier ===
          counterpartyConnectionIdentifier &&
        previous.counterpartyPrefix === counterpartyPrefix &&
        previous.clientIdentifier === clientIdentifier &&
        previous.counterpartyClientIdentifier ===
          counterpartyClientIdentifier))
```

where `previous` is the initialized connection end.

Recommendation

Add a mechanism in both the specification, and the implementation to deal with mismatched parameters.

Note At the time of writing this report, the following issues were opened to address this problem in the case when the connection (channel) identifiers do not match: Cosmos-ICS #491 and Cosmos-SDK #7870.

IF-IBC-12

Non-atomicity of operations

Severity: Informative
Type: Bad software practice / lack of documentation
Difficulty: Easy
Involved artifacts: ICS 20 Specification, bank/spec, ibc/types/coin.go, ibc/applications/transfer.
Issue: Cosmos SDK #8030

Description

In multiple places throughout the Cosmos-SDK, functions contain several sub-calls that may fail due to various reasons. At the same time, the sub-calls update the shared state, and the functions do not backtrack the state changes done by the first sub-calls if one of the subsequent sub-calls fails.

Consider this example of SubtractCoins():

```
for _, coin := range amt {
    balance := k.GetBalance(ctx, addr, coin.Denom)
    locked := sdk.NewCoin(coin.Denom, lockedCoins.AmountOf(coin.Denom))
    spendable := balance.Sub(locked)
    _, hasNeg := sdk.Coins{spendable}.SafeSub(sdk.Coins{coin})
    if hasNeg {
        return sdkerrors.Wrapf(sdkerrors.ErrInsufficientFunds, "%s is smaller than %s",
    }
    newBalance := balance.Sub(coin)
    err := k.SetBalance(ctx, addr, newBalance)
    if err != nil {
        return err
    }
}
```

The function iterates over the set of coins and for each coin checks whether enough coins are available for the current denomination. The balance is updated for each coin as the loop progresses. Consider the scenario where the balance for the first coin is updated successfully, but for the

second coin setting of the balance fails because of the negative amount of coins. The function returns the error, but the balance for the first coin remains updated in the shared state.

This violates one of the most basic assumptions of function atomicity, namely that either

1. the function updates the state and succeeds, or

2. the function returns an error, but the shared state is unmodified.

We have found multiple occasions of such non-atomic functions; here are some, besides the example above:

Bank:

- AddCoins similar issue with the difference it panics when overflow happens for some denomination.
- SendCoins first subtracts from the sender account and then adds to the receiver. If subtract is successful and add fails the state is changed.
- DelegateCoins: delegation is processed denomination by denomination: the insufficient funds is check inside the loop allowing state updates even when an error is reached.
- UndelegateCoins similar scenario.
- BurnCoins and MintCoins use SubtractCoins and AddCoins.

ICS20:

- OnRecvPacket: first minting then moving into the account. Each step modifies the bank state. Each step potentially reaches errors. Therefore it is not an atomic write to the bank as given in the spec.
- SendTransfer: similar to `OnRecvPacket`, first SendCoinsFromAccountToModule() is called, that modifies the state, and after that, if BurnCoins() fails, the error is returned, but the state is left modified.

The problem is that **all above functions have implicit assumption on the behavior of the caller**. This implicit assumption is that whenever any such function returns an error, the only correct behavior is to propagate this error up the stack.

While this assumption seems indeed to be satisfied by the present Cosmos SDK codebase (with one exception, see the problem scenario below), it is not documented anywhere in the Cosmos SDK developer documentation. There are only hints to this in the form similar to this paragraph in the Cosmos SDK handler documentation:

> The Context contains all the necessary information needed to process the msg, as well as a cache-wrapped copy of the latest state. If the msg is succesfully processed,

the modified version of the temporary state contained in the ctx will be written to the main state.

Such hints do not constitute enough developer guidance to avoid introducing severe bugs, especially for Cosmos SDK newcomers.

Problem Scenarios

We have found one particular place where this non-atomicity has almost led to the real bug. Namely, in ICS20 OnRecvPacket we have two consecutive operations, minting and sending.

```
// mint new tokens if the source of the transfer is the same chain
if err := k.bankKeeper.MintCoins(
    ctx, types.ModuleName, sdk.NewCoins(voucher),
); err != nil {
    return err
}


// send to receiver
if err := k.bankKeeper.SendCoinsFromModuleToAccount(
    ctx, types.ModuleName, receiver, sdk.NewCoins(voucher),
); err != nil {
    return err
}
```

If the minting succeeds, but the sending fails, the function returns an error, while the funds are moved to the module account.

This is how this function is called in applications/transfer/module.go:

```
err := am.keeper.OnRecvPacket(ctx, packet, data)
if err != nil {
    acknowledgement = channeltypes.NewErrorAcknowledgement(err.Error())
}
```

We see that the error is turned into a negative acknowledgement. **If sending of the coins above was to fail with an error, then the negative acknowledgement would be sent, but also the funds would be silently moved to the module account under the hood.** We have carefully examined the code of `SendCoinsFromModuleToAccount`, and found out that

its current implementation can only panic, e.g. when the receiver account overflows. But if there was a possibility for it to return an error this scenario would constitute a real problem. Please note that these two functions are probably written by two different developers, and it would be perfectly legitimate for `SendCoinsFromModuleToAccount` to return an error – this is how close it comes to being a real bug. Please see also the finding IF-IBC-01 for more details on this issue.

Recommendation

- **Short term**: properly **explain in the developer documentation the implicit assumption** of propagating all returned errors up the stack , as well as in the inline documentation for all functions exhibiting non-atomic behavior.
- **Long term**: one of the following needs to be done:
  - either **make the SDK functions atomic** as described above;
  - or **introduce a separate explicit step** for handlers, say `CommitState`, that the handler will need to call to write state changes to the store.

IF-IBC-13

ICS20 - Model and Model-based Tests for Token Transfer

Severity: Informative
Type: Set of new artifacts to facilitate rigorous testing
Involved original artifacts: ICS 20 Specification, ibc/applications/transfer.
Issues / PRs: Cosmos SDK #8120, Cosmos SDK #8145

Description

The set of developed artifacts covers the efforts of creating a preliminary model and a set of model-based tests for ICS-20 token transfer. While this work is of an exploratory nature, it already helped to catch the bug in the Go implementation; see Cosmos SDK #8120.

Considering ICS-20 Token Transfer a critical component of the IBC infrastructure, we have invested some time into formally modeling and verifying it using TLA+ and Apalache model checker, and creating the set of tests based on this model. Below we describe the main components of the developed artifact.

The TLA+ model of ICS-20 Token Transfer relay functions

The model in its entirety can be found here. Our TLA+ model is based both on the specification as well as on the implementation. The main file of the TLA+ model is relay.tla. For all main relay functions (`SendTransfer`, `OnRecvPacket`, `OnTimeoutPacket`, `OnAcknowledgementPacket`) it contains pre- and post-conditions of those functions. Based on those, it is possible to generate execution sequences of abstract "calls" to this functions, either successful or not, and to record those executions in the execution history. The history is later used to construct tests for the implementation in Go.

As an example we provide here a TLA+ precondition for the `OnRecvPacket` function:

```
OnRecvPacketPre(packet) ==
  LET data == packet.data
      trace == data.denomTrace
      denom == GetDenom(trace)
      amount == data.amount
```

```
  IN
  /\ WellFormedPacket(packet)
  /\ IsValidRecvChannel(packet)
  /\ IsValidDenomTrace(trace)
  /\ amount > 0
     \* if there is no receiver account, it is created by the bank
  /\ data.receiver /= NullId
  /\ IsSource(packet) =>
       LET escrow == GetDestEscrowAccount(packet) IN
       LET denomTrace == ReduceDenomTrace(trace) IN
           /\ <<escrow, denomTrace>> \in DOMAIN bank
           /\ bank[escrow, denomTrace] >= amount
```

Model-based tests for relay functions

The above TLA+ model is complemented with a set of model-based tests, that can be found in relay_tests.tla. Model-based tests are simple TLA+ assertions that describe desired executions. Here is a simple example:

```
TestUnescrowTokens ==
  \E s \in DOMAIN history :
     /\ history[s].handler = "OnRecvPacket"
     /\ history[s].error = FALSE
     /\ IsSource(history[s].packet)
```

This test requires an existence of a computation history state such that: * `OnRecvPacket` is called in this state; * the call is successful; * The received packet originated from the receiving chain.

When the model contains three chains, A, B, C, such that A is connected to B, and B is connected to C, this combination of conditions forces the model checker to generate an example execution consisting of three steps (we are on chain B): 1. Some tokens are received from A to B (`OnRecvPacket` on chain B); 2. Some of the received tokens are sent from B to C (`SendTransfer` on chain B); 3. Some of the previosly sent tokens are received back from C to B, and unescrowed there. (`OnRecvPacket` on chain B).

The model together with the test is executed by our model checker Apalache. The example execution is recorded by the model checker as a so-called *counterexample*.

Transformation specification

As a counterexample produced by the model checker contains an arbitrary syntax tree of TLA+, this needs to be translated into a machine-readable form, that can be parsed by the test driver. This translation is performed by another software component, Jsonatr (JSON Arrifact Translator). The transformation is described by a transformation specification.

Model-based test driver

Finally, the transformed model execution is executed by the test driver mbt_relay_test.go, which is a simple Go component, integrated into the IBC testing framework. The test driver does the following: 1. Sets up the test environment, consisting of the necessary number of chains and their connections; 2. Deserializes the transformed model execution; 3. For each step of the execution, calls the corresponding relay function of the token transfer module; 4. Checks the returned result and the changes in the chain bank module are as expected by the model execution.

Recommendation

Our model-based tests for the token transfer module have been successfully merged into Cosmos-SDK (see the PR Cosmos-SDK #8145), and already helped to catch a real implementation bug in a complicated 3-chain scenario; see IF-IBC-14.

At the same time, only a very limited number of tests are contained there. We recommend the Cosmos-SDK developers to master the model-based testing methodology, and to extend their test suite with more model-based tests. This should help both to reduce the testing efforts from the developers, and to increase the coverage of complicated scenarios.

IF-IBC-14

ICS20 - Panic on receiving multi-chain denominations

Severity: High
Type: Implementation bug
Difficulty: Low
Involved artifacts: applications/transfer/keeper/relay.go:OnRecvPacket(), model-based tests for token transfer
Issues / PRs: Cosmos-SDK #8120, Cosmos-SDK #8119

Description

In the following code fragment of applications/transfer/keeper/relay.go:OnRecvPacket(), it is assumed that the denomination trace can contain at most two components, i.e. the unprefixed denomination is a native token:

```
if types.ReceiverChainIsSource(packet.GetSourcePort(),
     packet.GetSourceChannel(), data.Denom) {
    // sender chain is not the source, unescrow tokens
    // remove prefix added by sender chain
    voucherPrefix := types.GetDenomPrefix(packet.GetSourcePort(),
      packet.GetSourceChannel())
    unprefixedDenom := data.Denom[len(voucherPrefix):]
    token := sdk.NewCoin(unprefixedDenom, sdk.NewIntFromUint64(data.Amount))
```

This assumption was valid for all Cosmos-SDK tests for token transfer present at the time of our audit, in particular for the quite extensive handler_test.go.

Nevertheless, the following simple TLA+ test, that was executed as part of our model-based testing efforts for token transfer module, generated an execution with a token transfer crossing 3 chains:

```
TestUnescrowTokens ==
  \E s \in DOMAIN history :
     /\ IsSource(history[s].packet)
     /\ history[s].handler = "OnRecvPacket"
     /\ history[s].error = FALSE
```

This 3-chain token transfer resulted in the receiving denomination being a non-native token, and led to the panic in the implementation; see details in the issue Cosmos-SDK #8120.

Problem Scenarios

Consider the following scenario, involving three chains: A, B, and C. Transfer channel from A to B has as id `channel-0` for both channel ends, and transfer channel from B to C has as ids for channel ends `channel-1` and `channel-0` respectively. We are interested in the token transfer module of chain B, and consider all steps relative to chain B. The following steps are generated from the TLA+ test above:

0. Initialization: bank of B is empty.
1. Receive 5 atoms from chain A to account a3
   - transferred denomination: `atom`
   - expected bank state:
     - `<a3, transfer/channel-0/atom> = 5`
2. Send 3 atoms from account a3 on chain B to chain C (funds are moved to the escrow account)
   - transferred denomination: `transfer/channel-0/atom`
   - expected bank state:
     - `<a3, transfer/channel-0/atom> = 2`
     - `<transfer/channel-1, transfer/channel-0/atom> = 3`
3. Receive 1 atom from chain C to account a1 on chain B (funds are moved from the escrow account)
   - transferred denomination: `transfer/channel-0/transfer/channel-0/atom`
   - expected bank state:
     - `<a3, transfer/channel-0/atom> = 2`
     - `<transfer/channel-1, transfer/channel-0/atom> = 2`
     - `<a1, transfer/channel-0/atom> = 1`

As can be seen in the step 3 above, the unprefixed denomination is not native, which leads to the implementation panic.

Recommendation

The problem has been promptly fixed by the developers in this PR: Cosmos-SDK #8119. The fix is simple, and consists of hashing of the received denomination.

The developers have also promptly updated the hand-written test in order to test the 3-chain scenario above, see the file changes. It is worth comparing the length of the modifications to the hand-written test that were necessary to cover the 3-chain scenario with the conciseness of the TLA+ test above.

While our model-based tests for the token transfer module, including the TLA+ test above, have been successfully merged into Cosmos-SDK (see the PR Cosmos-SDK #8145), only a very limited number of tests are contained there. We recommend the Cosmos-SDK developers to master the model-based testing methodology (see IF-IBC-13 for a short description), and to extend their test suite with more model-based tests. This should help both to reduce the testing efforts from the developers, and to increase the coverage of complicated scenarios.