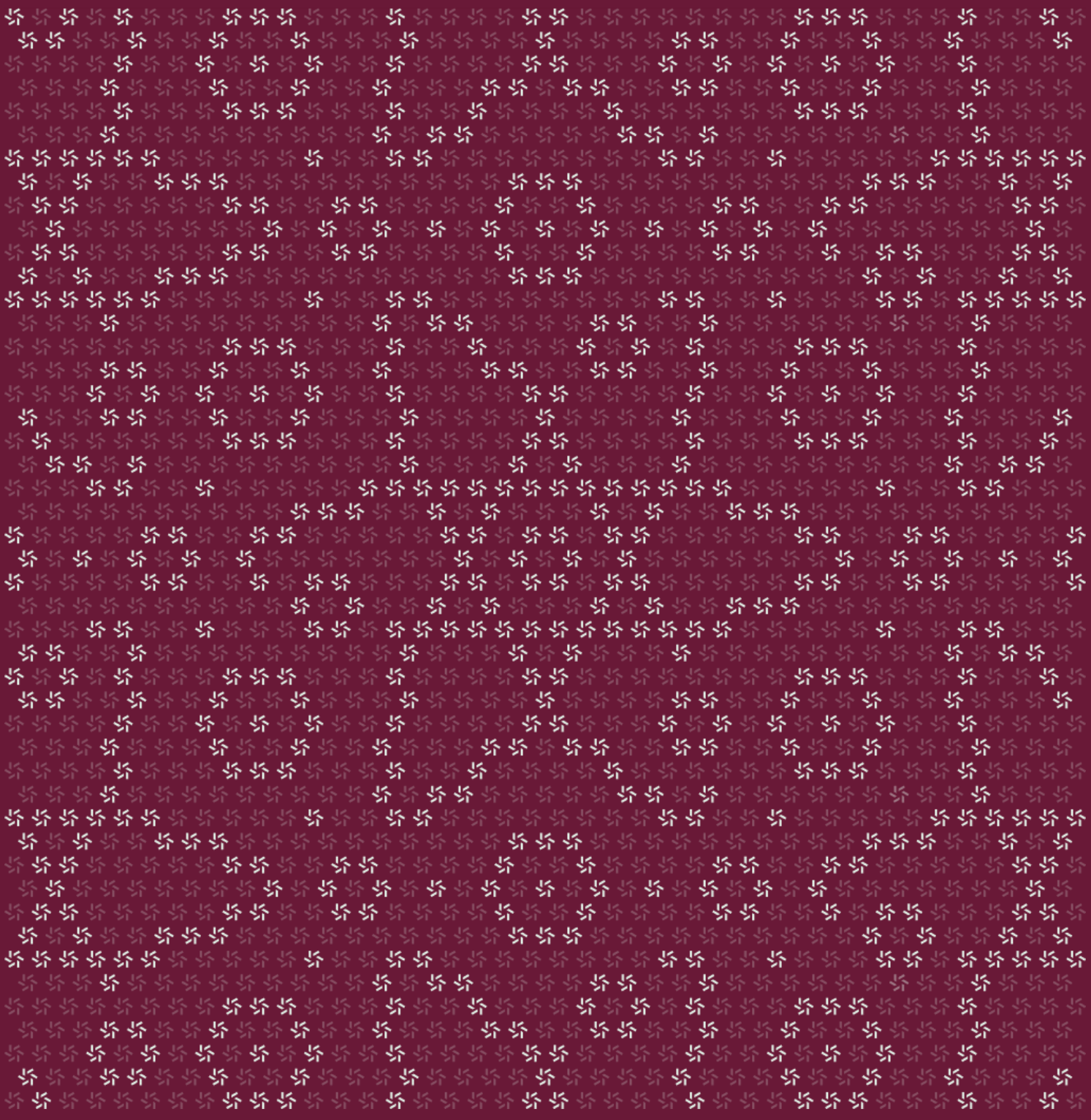
Zellic

Prepared for
Henry de Valence
Penumbra Labs, Inc

Prepared by
Syed Faraz Abrar
William Bowling
Avraham Weinstock
Zellic

April 25, 2024

Penumbra IBC
Blockchain Security Assessment



Contents

DRAFT

DRAFT

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

DRAFT

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Penumbra Labs, Inc from March 4th to April 17th, 2024. During this engagement, Zellic reviewed Penumbra IBC's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does `penumbra-ibc` implement the ICS-{002,003,004,007} protocols consistently with `ibc-go` and with the specification?
- Does `penumbra-ibc` correctly make use of ICS-023 vector commitments?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The security of the implementation of the underlying cryptographic primitives, including ics23 vector commitments
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

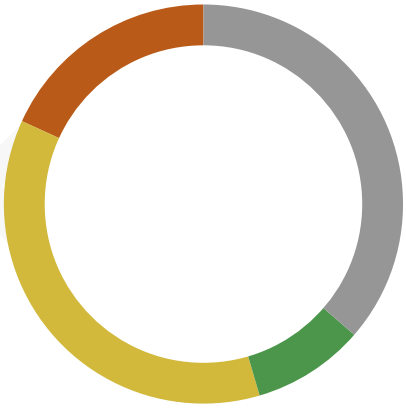
During our assessment on the scoped Penumbra IBC crates, we discovered 11 findings. No critical issues were found. Two findings were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Penumbra Labs, Inc's benefit in the Discussion section (4. ↗) at the end of the document.

DRAFT

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	4
Low	1
Informational	4



DRAFT

2. Introduction

2.1. About Penumbra IBC

Penumbra Labs, Inc contributed the following description of Penumbra IBC:

> Penumbra is a fully private proof-of-stake network and decentralized exchange.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the crates.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

DRAFT

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped crates itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

DRAFT

2.3. Scope

The engagement involved a review of the following targets:

Penumbra IBC Crates

Repository	https://github.com/penumbra-zone/penumbra ↗
Version	penumbra: `c462e9a6cc86a37d42a9d75f1ab9ceb0e3789f77`
Program	crates/core/component/ibc/src/*
Type	Rust
Platform	Cosmos-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of 6.5 person-weeks. The assessment was conducted over the course of six calendar weeks.

DRAFT

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar
Engineer
faith@zellic.io

William Bowling
Engineer
vakzz@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 4, 2024	Start of primary review period
April 17, 2024	End of primary review period
TBD	Closing call

DRAFT

3. Detailed Findings

3.1. False negative for Timeouts on ordered channels

Target	crates/core/component/ibc		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When handling `MsgTimeout` on ordered channels, `penumbra-ibc` uses `!=` on sequence numbers here ↗, so it will only accept proof that the most recent packet timed out.

```
if channel.ordering == ChannelOrder::Ordered {
    // ordered channel: check that packet has not been received
    if self.next_seq_recv_on_b != self.packet.sequence {
        anyhow::bail!("packet sequence number does not match");
    }

    // in the case of a timed-out ordered packet, the counterparty should have
    // committed the next sequence number to their state
    state
        .verify_packet_timeout_proof::<HI>(&connection, self)
        .await
        .context("failed to verify packet timeout proof")?;
} else {
```

The corresponding check in `ibc-go` here ↗ uses `>`, so it will accept evidence of any past packet having timed out.

```
switch channel.Ordering {
case types.ORDERED:
    // check that packet has not been received
    if nextSequenceRecv > packet.GetSequence() {
        return errorsmod.Wrapf(
            types.ErrPacketReceived,
            "packet already received, next sequence receive > packet sequence
    (%d > %d)", nextSequenceRecv, packet.GetSequence(),
        )
    }

    // check that the recv sequence is as claimed
    err = k.connectionKeeper.VerifyNextSequenceRecv(
```

```
        ctx, connectionEnd, proofHeight, proof,
        packet.GetDestPort(), packet.GetDestChannel(), nextSequenceRecv,
    )
```

Impact

If other implementations that are tested against `ibc-go` send `MsgTimeout` for a packet other than the latest when multiple in-flight packets time out over an ordered channel, `penumbra-ibc` will not acknowledge those, resulting in the timed-out channel not being closed (potentially locking ICS-20 escrowed funds, though this might be fixable by manually sending a `MsgTimeout` for the latest applicable packet on that channel).

Recommendations

Enforce that the packet's sequence number is less than or equal to the current expected sequence number when handling a time-out.

```
     // ordered channel: check that packet has not been received
- if self.next_seq_recv_on_b != self.packet.sequence {
+ if self.next_seq_recv_on_b > self.packet.sequence {
         anyhow::bail!("packet sequence number does not match");
     }
```

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit 269ab197↗.

3.2. Missing expiration check in `UpgradeClient`

Target	src/component/msg_handler/upgrade_client.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The handling of `UpgradeClient` by `ibc-go` checks both that the client is not frozen and not expired. The handling of `UpgradeClient` by `penumbra-ibc` only checks that the client is not frozen.

The `UpgradeClient` function calls `GetClientStatus`.

```
func (k Keeper) UpgradeClient(
    ctx sdk.Context,
    clientID string,
    upgradedClient, upgradedConsState, upgradeClientProof,
    upgradeConsensusStateProof []byte,
) error {
    if status := k.GetClientStatus(ctx, clientID); status != exported.Active {
        return errorsmod.Wrapf(types.ErrClientNotActive, "cannot upgrade
    client (%s) with status %s", clientID, status)
    }
    // ...
```

The `GetClientStatus` function calls `ClientState.Status`.

```
func (k Keeper) GetClientStatus(ctx sdk.Context, clientID string)
    exported.Status {
    // ...
    return clientModule.Status(ctx, clientID)
}
```

The `ClientState.Status` function includes a check for expiration.

```
func (cs ClientState) Status(
    ctx sdk.Context,
    clientStore storetypes.KVStore,
    cdc codec.BinaryCodec,
) exported.Status {
```

```
    if !cs.FrozenHeight.IsZero() {
        return exported.Frozen
    }

    // get latest consensus state from clientStore to check for expiry
    consState, found := GetConsensusState(clientStore, cdc, cs.LatestHeight)
    if !found {
        // if the client state does not have an associated consensus state for
    its latest height
        // then it must be expired
        return exported.Expired
    }

    if cs.IsExpired(consState.Timestamp, ctx.BlockTime()) {
        return exported.Expired
    }

    return exported.Active
}
```

The `verify_client_upgrade_proof` function only checks ↗ if the client is frozen.

```
// check if the client is frozen
if trusted_client_state.is_frozen() {
    anyhow::bail!("client is frozen");
}
```

Impact

This allows a chain to use an `UpgradeClient` message to re-activate an expired client state.

Recommendations

Check that the trusted client state is nonexpired in `ClientUpgradeProofVerifier::verify_client_upgrade_proof`.

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit `9f4abd0e` ↗.

3.3. Upgrade-path length hardcoded to two

Target	src/component/proof_verification.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Whereas `ibc-go` uses all but the last element of `UpgradePath` for verifying the client and consensus state proofs when processing an `UpgradeClient` message, `penumbra-ibc` only uses the first element of `upgrade_path`, which is equivalent to an assumption that the path length is always two.

The ibc-go library uses ↗ `constructUpgradeClientMerklePath` and `constructUpgradeConsStateMerklePath` to construct paths for the Merkle proofs.

```
// Verify client proof
bz, err := cdc.MarshalInterface(tmUpgradeClient.ZeroCustomFields())
if err != nil {
    return errorsmod.Wrapf(clienttypes.ErrInvalidClient, "could not marshal
    client state: %v", err)
}
// construct clientState Merkle path
upgradeClientPath := constructUpgradeClientMerklePath(cs.UpgradePath,
    lastHeight)
if err := merkleProofClient.VerifyMembership(cs.ProofSpecs,
    consState.GetRoot(), upgradeClientPath, bz); err != nil {
    return errorsmod.Wrapf(err, "client state proof failed. Path: %s",
    upgradeClientPath.GetKeyPath())
}

// Verify consensus state proof
bz, err = cdc.MarshalInterface(upgradedConsState)
if err != nil {
    return errorsmod.Wrapf(clienttypes.ErrInvalidConsensus, "could not marshal
    consensus state: %v", err)
}
// construct consensus state Merkle path
upgradeConsStatePath := constructUpgradeConsStateMerklePath(cs.UpgradePath,
    lastHeight)
if err := merkleProofConsState.VerifyMembership(cs.ProofSpecs,
    consState.GetRoot(), upgradeConsStatePath, bz); err != nil {
    return errorsmod.Wrapf(err, "consensus state proof failed. Path: %s",
```

```
    upgradeConsStatePath.GetKeyPath())
}
```

The `constructUpgradeClientMerklePath` function uses ↗ all but the last element of `upgradePath` as a prefix of `clientPath`.

```
// construct MerklePath for the committed client from upgradePath
func constructUpgradeClientMerklePath(upgradePath []string, lastHeight
    exported.Height) commitmenttypes.MerklePath {
    // copy all elements from upgradePath except final element
    clientPath := make([]string, len(upgradePath)-1)
    copy(clientPath, upgradePath)

    // append lastHeight and `upgradedClient` to last key of upgradePath and
    use as lastKey of clientPath
    // this will create the IAVL key that is used to store client in upgrade
    store
    lastKey := upgradePath[len(upgradePath)-1]
    appendedKey := fmt.Sprintf("%s/%d/%s", lastKey,
    lastHeight.GetRevisionHeight(), upgradetypes.KeyUpgradedClient)

    clientPath = append(clientPath, appendedKey)
    return commitmenttypes.NewMerklePath(clientPath...)
}
```

The `verify_client_upgrade_proof` function only uses ↗ `upgrade_path[0]` in `upgrade_path_prefix`.

```
let upgrade_path_prefix =
    MerklePrefix::try_from(upgrade_path[0].clone().into_bytes())
    .map_err(|_| {
        anyhow::anyhow!("couldn't create commitment prefix from client upgrade
    path")
    })?;

// check if the client is frozen
if trusted_client_state.is_frozen() {
    anyhow::bail!("client is frozen");
}

// get the stored consensus state for the counterparty
let trusted_consensus_state = self
    .get_verified_consensus_state(&trusted_client_state.latest_height(),
    client_id)
    .await?;
```

```
verify_merkle_proof(
    &trusted_client_state.proof_specs,
    &upgrade_path_prefix,
    client_state_proof,
    &trusted_consensus_state.root,
    ClientUpgradePath::UpgradedClientState(
        trusted_client_state.latest_height().revision_height(),
    ),
    upgraded_tm_client_state.encode_to_vec(),
)?;
```

Impact

If the upgrade path is ever longer than two, a malicious chain may be able to include different client and consensus state proofs at different paths, upgrading its state to different values on `penumbra-ibc` and `ibc-go` chains.

Recommendations

Use all but the last element of `upgrade_path` instead of `upgrade_path[0]` in `ClientUpgradeProofVerifier::verify_client_upgrade_proof`.

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit 3c38d2a3.

DRAFT

3.4. Handshake state machines too permissive when processing Acks

Target	crates/core/component/ibc		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Both `ConnectionOpenAck` and `ChannelOpenAck` permit the party processing the message to have their channel in `State::TryOpen` in addition to `State::Init`.

The `verify_previous_connection` function permits `State::TryOpen` in `ConnectionOpenAck`.

```
let state_is_consistent = connection.state_matches(&State::Init)
    && connection.versions.contains(&msg.version)
    || connection.state_matches(&State::TryOpen)
        && connection.versions.get(0).eq(&Some(&msg.version));
```

The `channel_state_is_correct` function permits `ChannelState::TryOpen` in `ChannelOpenAck`.

```
fn channel_state_is_correct(channel: &ChannelEnd) -> anyhow::Result<()> {
    if channel.state == ChannelState::Init || channel.state
    == ChannelState::TryOpen {
        Ok(())
    } else {
        Err(anyhow::anyhow!("channel is not in the correct state"))
    }
}
```

The `ibc-go` library requires the connections/channels to be strictly in state `INIT`.

The `ConnOpenAck` function enforces the connection's state to be `types.INIT`.

```
// verify the previously set connection state
if connection.State != types.INIT {
    return errorsmod.Wrapf(
        types.ErrInvalidConnectionState,
        "connection state is not INIT (got %s)", connection.State.String(),
    )
}
```

The ChanOpenAck function enforces the channel's state to be `types.INIT`.

```
if channel.State != types.INIT {
    return errorsmod.Wrapf(types.ErrInvalidChannelState, "channel state should
    be INIT (got %s)", channel.State.String())
}
```

Impact

It may be possible for a malicious chain to swap roles partway through the handshake (in the following, A is an attacker-controlled chain and B is a `penumbra-ibc` chain initiating a connection with it):

- B sends `ConnectionOpenInit` to A; state is (`Init`, `None`).
- A sends `ConnectionOpenTry` to B; state is (`Init`, `TryOpen`).
- B sends `ConnectionOpenAck` to A; state would normally be in (`Open`, `TryOpen`), but A can deviate here and store `TryOpen`, resulting in state (`TryOpen`, `TryOpen`).
- A deviates from the protocol and sends `ConnectionOpenAck` to B (instead of sending `ConnectionOpenConfirm`). If B is an `ibc-go` node, B rejects this since it is in `TRYOPEN` state and not `INIT` state, but on `penumbra-ibc`, this check succeeds. At this point, to pass the connection -erkle proof check, A needs to prove that their channel is in `TryOpen` state, which they can do if they deviated above. At this point, the client and consensus-state proofs that B is processing refer to A's representation of B's state. If this step succeeds, (A, B) are in state (`TryOpen`, `Open`).
- B sends `ConnectionOpenConfirm` to A (at this point the roles have reversed), and the resulting state is (`Open`, `Open`).

This may result in not all relevant proofs being checked in both directions or IDs stored in B being chosen by A.

Recommendations

Enforce that `connection.state/channel.state` are only in `State::Init` in `MsgConnectionOpenAck::try_execute/ChannelOpenAck::try_execute`, respectively.

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit `c4e51bc8`.

3.5. Zero out custom fields on client update

Target	src/component/msg_handler/upgrade_client.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When a client upgrade is performed, the `verify_client_upgrade_proof` function is called to ensure the counterparty has committed the new client state:

```
let upgraded_client_state_tm =
    TendermintClientState::try_from(self.client_state.clone())
            .context("client state is not a Tendermint client state")?;
// [...snip...]
state
    .verify_client_upgrade_proof(
        &self.client_id,
        &proof_client_state,
        &proof_consensus_state,
        upgraded_consensus_state_tm.clone(),
        upgraded_client_state_tm.clone(),
    )
    .await?;
```

The corresponding check in `ibc-go` here ↗ used a zeroed-out client state to verify the proof:

```
// Verify client proof
bz, err := cdc.MarshalInterface(tmUpgradeClient.ZeroCustomFields())
if err != nil {
    return errorsmod.Wrapf(clienttypes.ErrInvalidClient, "could not marshal
    client state: %v", err)
}
// construct clientState Merkle path
upgradeClientPath := constructUpgradeClientMerklePath(cs.UpgradePath,
    lastHeight)
if err := merkleProofClient.VerifyMembership(cs.ProofSpecs,
    consState.GetRoot(), upgradeClientPath, bz); err != nil {
    return errorsmod.Wrapf(err, "client state proof failed. Path: %s",
    upgradeClientPath.GetKeyPath())
}
```

It also uses the zeroed-out client state when committing the new client state here ↗:

```
// ScheduleIBCSoftwareUpgrade schedules an upgrade for the IBC client.
func (k Keeper) ScheduleIBCSoftwareUpgrade(ctx sdk.Context, plan
    upgradetypes.Plan, upgradedClientState exported.ClientState) error {
    // zero out any custom fields before setting
    cs, ok := upgradedClientState.(*ibctm.ClientState)
    if !ok {
        return errorsmod.Wrapf(types.ErrInvalidClientType, "expected: %T, got:
    %T", &ibctm.ClientState{}, upgradedClientState)
    }

    cs = cs.ZeroCustomFields()
    bz, err := types.MarshalClientState(k.cdc, cs)
    if err != nil {
        return errorsmod.Wrap(err, "could not marshal UpgradedClientState")
    }

    if err := k.upgradeKeeper.ScheduleUpgrade(ctx, plan); err != nil {
        return err
    }

    // sets the new upgraded client last height committed on this chain at
    plan.Height,
    // since the chain will panic at plan.Height and new chain will resume at
    plan.Height
    if err = k.upgradeKeeper.SetUpgradedClient(ctx, plan.Height, bz); err
    != nil {
        return err
    }
```

Impact

If the counterparty performing the update is using the `ibc-go` implementation, Penumbra will fail to verify the client upgrade proof as it will be different to the zeroed-out client state used by `ibc-go`, resulting in a failed upgrade.

Recommendations

All customizable fields in the client state should be zeroed out before verifying the client upgrade proof, as done in `ibc-go`:

```
func (cs ClientState) ZeroCustomFields() *ClientState {
    // copy over all chain-specified fields
```

```
    // and leave custom fields empty
    return &ClientState{
        ChainId: cs.ChainId,
        UnbondingPeriod: cs.UnbondingPeriod,
        LatestHeight: cs.LatestHeight,
        ProofSpecs: cs.ProofSpecs,
        UpgradePath: cs.UpgradePath,
    }
}
```

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit 1851fe04 ↗.

DRAFT

3.6. Chain ID parsing incorrectly accepts newlines

Target	crates/core/component/ibc		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

When parsing chain IDs, `ibc-types` uses a more lenient regex ↗ than `ibc-go`'s regex ↗.

```
var IsRevisionFormat = regexp.MustCompile(`^.*[^\n-]-{1}[1-9][0-
    9]*$`).MatchString
```

```
pub fn is_epoch_format(chain_id: &str) -> bool {
    let re = safe_regex::regex!(br".*[^-]-[1-9][0-9]*");
    re.is_match(chain_id.as_bytes())
}
```

For example, `ibc-go` will consider the chain ID `\n-foo-1` to have revision number 0, whereas `penumbra-ibc` will consider it to have revision number 1.

Impact

This likely does not have any security impact, but it is a deviation from the specification and can potentially lead to an issue in the future; see Discussion point 4.1. ↗.

Recommendations

Include a newline (and possibly anchors) in `ibc-types`'s version of the regex.

```
 pub fn is_epoch_format(chain_id: &str) -> bool {
- let re = safe_regex::regex!(br".*[^-]-[1-9][0-9]*");
+ let re = safe_regex::regex!(br"^.*[^\n-]-[1-9][0-9]*$");
     re.is_match(chain_id.as_bytes())
 }
```

The anchors are not strictly required since `safe_regex::IsMatch::is_match` checks if the whole string matches, in contrast to Go's `Regexp.MatchString`, which checks if any substring matches;

the behavioral difference is in the lack of `\n` in the character class.

Remediation

TBD

DRAFT

3.7. Missing check in `ClientState::new`

Target	ibc-types/crates/ibc-types-lightclients-tendermint/src/client_state.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `ClientState::new` function in `ibc-types` shares most of its checks with `ibc-go`'s `ClientState.Validate` but omits the check that `chain_id` is nonempty, and none of `ChainId`'s `From`/`FromStr` trait implementations contain this check. This will result in `penumbra-ibc` nodes accepting chain IDs that `ibc-go` nodes will reject (though when referring to the chain ID `""`, it will reserialize it as `"-0"`, which `ibc-go` will accept).

The `ibc-go` library checks that the chain ID is nonempty.

```
if strings.TrimSpace(cs.ChainId) == "" {
    return errorsmod.Wrap(ErrInvalidChainID, "chain id cannot be empty string")
}
```

The `ChainId::new` function in `ibc-types` omits the corresponding check.

```
pub fn from_string(id: &str) -> Self {
    let version = if Self::is_epoch_format(id) {
        Self::chain_version(id)
    } else {
        0
    };

    Self {
        id: id.to_string(),
        version,
    }
}
```

Impact

This likely does not have any security impact, but it is a deviation from the specification and can potentially lead to an issue in the future; see Discussion point 4.1. ↗.

Recommendations

Use fallible conversions from `String`/`&str` to `ChainId` and reject the empty string.

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit `73da5ee5` ↗

DRAFT

3.8. Frozen client height mismatches with `ibc-go` spec

Target	crates/core/component/ibc		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

Clients that are frozen are always frozen at a specific height. This height has a revision number of 0 and a revision height of 1. This can be seen in the IBC specifications here ↗ and in `ibc-go` here ↗.

Penumbra instead uses the current Tendermint block header's height as the frozen height, as seen here ↗.

Impact

This likely does not have any security impact, but it is a deviation from the specification and can potentially lead to an issue in the future; see Discussion point 4.1. ↗.

Recommendations

Use a revision number of 0 and height of 1 instead of the current Tendermint block header's height for frozen clients.

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit 9c463357 ↗.

DRAFT

3.9. Revision number not in state

Target	crates/core/component/ibc		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

In `ibc-go`, the revision number parsed from the chain ID as part of the processed height is stored at `"consensusStates/{height}/processedHeight"`; in contrast, `penumbra-ibc` hardcodes a 0 revision number in the processed height and stores it at `"ibc/-clients/{client_id}/processedHeights/{height}"`.

The `GetHeight` function ↗ in `ibc-go` parses the revision number from the chain ID.

```
func (h Header) GetHeight() exported.Height {
    revision := clienttypes.ParseChainID(h.Header.ChainID)
    return clienttypes.NewHeight(revision, uint64(h.Header.Height))
}
```

The `UpdateState` function ↗ calls `setConsensusMetadata`, which writes to `"consensusStates/{height}/processedHeight"`.

```
func (cs ClientState) UpdateState(ctx sdk.Context, cdc codec.BinaryCodec,
    clientStore storetypes.KVStore, clientMsg exported.ClientMessage)
    []exported.Height {
    // ...
    setConsensusMetadata(ctx, clientStore, header.GetHeight())
    // ...
}
```

The `put_verified_consensus_state` function ↗ uses a hardcoded 0 revision number in the stored `processedHeight`.

```
self.put(
    state_key::client_processed_heights(&client_id, &height),
    ibc_types::core::client::Height::new(0, current_height)?,
);
```

Impact

This likely does not have any security impact, but it is a deviation from the specification and can potentially lead to an issue in the future; see Discussion point 4.1. ↗.

Recommendations

Make the heights and paths used for `put_verified_consensus_state` consistent with `setConsensusMetadata`.

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit `94b7b15c` ↗.

DRAFT

3.10. Trailing hex digits in chain IDs may be treated as revisions

Target	crates/core/component/ibc		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

If `preserve_chain_id` is false when running `pd testnet generate`, the generated testnet will have a name ending in 8 hex nybbles, which can be parsed as a decimal approximately 2.3% of the time ($(\frac{10}{16})^8$), which may be interpreted as a revision.

This is the pd executable ↗'s generation of hex testnet names:

```
let chain_id = match preserve_chain_id {
    true =>
    chain_id.unwrap_or_else(|| env!("PD_LATEST_TESTNET_NAME").to_string()),
    false => {
        // If preserve_chain_id is false, we append a random suffix to avoid
    collisions
        let randomizer = OsRng.gen::<u32>();
        let chain_id =
            chain_id.unwrap_or_else(||
    env!("PD_LATEST_TESTNET_NAME").to_string());
        format!("{}-{}", chain_id, hex::encode(randomizer.to_le_bytes()))
    }
};
```

Impact

This likely does not have any security impact, but it is a deviation from the specification and can potentially lead to an issue in the future; see Discussion point 4.1. ↗.

Recommendations

Either append a `-0` to the chain ID or encode the random value as decimal (so that it is consistently treated like a revision).

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit df502ebc ↗.

DRAFT

3.11. Mismatch in upper bound for `TrustThreshold`

Target	src/component/ics02_validation.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

Tendermint's `TrustThresholdFraction::new` enforces that a trust threshold is in the interval $\left[\frac{1}{3}, 1\right]$. Also, `TrustThresholdFraction::new` is called from `ibc-types`'s `ClientState::new`, which shares most of its checks with `ibc-go`'s `ClientState.Validate`, and the corresponding check there is to `cometbft`'s `light.ValidateTrustLevel`, which also uses the interval $\left[\frac{1}{3}, 1\right]$.

Moreover, `penumbra-ibc` contains an additional check, `validate_trust_threshold`, which enforces that the trust threshold used when validating the counterparty's representation of Penumbra's state is in `ConnectionOpenTry` and `ConnectionOpenAck` is in the interval $\left[\frac{1}{3}, 1\right)$ (i.e., strictly less than one), which is more restrictive. The corresponding check in `ibc-go`'s `ConsensusHost.ValidateSelfClient` uses the same `light.ValidateTrustLevel` as `ClientState.Validate`.

The `TrustThresholdFraction::new` function ↗'s enforcement of the interval $\left[\frac{1}{3}, 1\right]$:

```
if numerator > denominator {
    return Err(Error::trust_threshold_too_large());
}
if denominator == 0 {
    return Err(Error::undefined_trust_threshold());
}
if 3 * numerator < denominator {
    return Err(Error::trust_threshold_too_small());
}
```

The `cometbft` library ↗'s enforcement of the interval $\left[\frac{1}{3}, 1\right]$:

```
func ValidateTrustLevel(lvl cmtmath.Fraction) error {
    if lvl.Numerator*3 < lvl.Denominator || // < 1/3
        lvl.Numerator > lvl.Denominator || // > 1
        lvl.Denominator == 0 {
        return fmt.Errorf("trustLevel must be within [1/3, 1], given %v", lvl)
    }
    return nil
```

```
}
```

This is the validate_trust_threshold function ↗'s enforcement of the interval $\left[\frac{1}{3}, 1\right)$:

```
fn validate_trust_threshold(trust_threshold: TrustThreshold) ->
    anyhow::Result<()> {
    if trust_threshold.denominator() == 0 {
        anyhow::bail!("trust threshold denominator cannot be zero");
    }

    if trust_threshold.numerator() * 3 < trust_threshold.denominator() {
        anyhow::bail!("trust threshold must be greater than 1/3");
    }

    if trust_threshold.numerator() >= trust_threshold.denominator() {
        anyhow::bail!("trust threshold must be strictly less than 1");
    }

    Ok(())
}
```

The ibc-go library ↗ uses `cometbft`'s `ValidateTrustLevel` in `ValidateSelfClient`.

```
if err := light.ValidateTrustLevel(tmClient.TrustLevel.ToTendermint()); err !=
    nil {
    return errorsmod.Wrapf(clienttypes.ErrInvalidClient, "trust-level invalid:
    %v", err)
}
```

Impact

If a `penumbra-ibc` node were configured to use a trust threshold of 1 (which is technically valid, but has liveness issues and may not provide additional security over a threshold of $\frac{2}{3}$), it would incorrectly not be able to open ICS-003 connections, since it would fail to recognize its own state on the counterparty's chain.

Recommendations

The `validate_trust_threshold` function should permit a `trust_threshold` of 1.

```
- if trust_threshold.numerator() >= trust_threshold.denominator() {
+ if trust_threshold.numerator() > trust_threshold.denominator() {
```

```
- anyhow::bail!("trust threshold must be strictly less than 1");
+ anyhow::bail!("trust threshold must be less than or equal to 1");
      }
```

Remediation

This issue has been acknowledged by Penumbra Labs, Inc, and a fix was implemented in commit 5e271118 ↗.

DRAFT

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Impact of provable store divergences not currently used in proofs

Several findings involve `penumbra-ibc` storing a different state than `ibc-go` does (either storing a non–byte-identical state at the same keys, storing values at different keys, or both).

While it does not seem like these can be amplified into a behavioral difference with the current implementations, this is a potential forwards-compatibility hazard if Merkle proofs of values at those paths will be needed by future protocols or if any existing protocol can depend on Merkle proofs with arbitrary keys.

4.2. ICS specification inconsistencies

There are a few places where the ICS specifications do not match either `penumbra-ibc` or `ibc-go` or are internally inconsistent. These are to be considered bugs in the specification itself, not bugs in `penumbra-ibc` or `ibc-go`.

ICS-007 specification omits the consensus-state Merkle proof

ICS-007's upgrade procedure ↗ only checks that the new client state is included in the previous consensus state, not that the new consensus state is included. In contrast, `penumbra-ibc` ↗ and `ibc-go` ↗ check both proofs.

ICS-007 specification is more permissive than implementations regarding upgrading frozen clients

When upgrading, ICS-007 permits ↗ a client frozen with a nonsentinel value to be upgraded if it were frozen in the future; neither `penumbra-ibc` ↗ nor `ibc-go` ↗ permit frozen clients to be upgraded at all.

Internal inconsistency in ICS-004

ICS-004's `recvPacket` function references `verifyPacketData` ↗, which seems to be an irrelevant reference to ICS-010. It seems likely that the intended reference is to ICS-003's `verifyPacketCommitment`, which is defined but unreferenced by the rest of the specification.

DRAFT

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the crates and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

Action: IbcRelay

The `IbcRelay` action encapsulates IBC messages that manage clients, channels, and connections.

Message: `CreateClient`

The `CreateClient` messages create a representation of a remote Tendermint light client with a specified initial consensus state; `client_state` must deserialize to a `ClientState` value, and `consensus_state` must deserialize to a `ConsensusState` value. The `ClientState` fields are checked to satisfy additional constraints (e.g., that the durations are nonzero, without which updates would be rejected, and that the trust level is between $\frac{1}{3}$ and 1, without which updates would be unsound).

```
pub struct MsgCreateClient {
    pub client_state: Any,
    pub consensus_state: Any,
    pub signer: String,
}

pub struct ClientState {
    pub chain_id: ChainId,
    pub trust_level: TrustThreshold,
    pub trusting_period: Duration,
    pub unbonding_period: Duration,
    pub max_clock_drift: Duration,
    pub latest_height: Height,
    pub proof_specs: Vec<ProofSpec>,
    pub upgrade_path: Vec<String>,
    pub allow_update: AllowUpdate,
    pub frozen_height: Option<Height>,
}

pub struct ConsensusState {
    pub timestamp: Time,
    pub root: MerkleRoot,
    pub next_validators_hash: Hash,
}
```

Message: `UpdateClient`

The `UpdateClient` messages update the state of an existing remote Tendermint light client; `client_message` must deserialize to a Tendermint `Header`, and the header must be valid according to the previous client and consensus states.

```
pub struct MsgUpdateClient {
    pub client_id: ClientId,
    pub client_message: Any,
    pub signer: String,
}

pub struct Header {
    pub signed_header: SignedHeader, // contains the commitment root
    pub validator_set: ValidatorSet, // the validator set that signed Header
    pub trusted_height: Height, // the height of a trusted header seen by
    client less than or equal to Header
    // TODO(thane): Rename this to trusted_next_validator_set?
    pub trusted_validator_set: ValidatorSet, // the last trusted validator set
    at trusted height
}
```

Message: `UpgradeClient`

The `UpgradeClient` messages upgrade an existing remote Tendermint light client to a new version; `client_state` must deserialize to a `ClientState` value, and `consensus_state` must deserialize to a `ConsensusState` value, as in `CreateClient`. The client must have opted into being upgradable by setting an `upgrade_path` in `CreateClient`, which is used as a path into the Merkle proofs to verify that the new replacement `client_state` and `consensus_state` were committed to by the most recently validated consensus state.

```
pub struct MsgUpgradeClient {
    // client unique identifier
    pub client_id: ClientId,
    // Upgraded client state
    pub client_state: Any,
    // Upgraded consensus state, only contains enough information
    // to serve as a basis of trust in update logic
    pub consensus_state: Any,
    // proof that old chain committed to new client
    pub proof_upgrade_client: RawMerkleProof,
    // proof that old chain committed to new consensus state
    pub proof_upgrade_consensus_state: RawMerkleProof,
    // signer address
```

```
    pub signer: String,
}
```

Message: `SubmitMisbehavior`

The `SubmitMisbehaviour` messages allow a node that notices certain forms of misbehavior (either producing two blocks for the same timestamp or including two blocks out of order) on a remote Tendermint client to alert other nodes, freezing their representations of the misbehaving node to prevent further divergence.

```
pub struct MsgSubmitMisbehaviour {
    /// client unique identifier
    pub client_id: ClientId,
    /// misbehaviour used for freezing the light client
    pub misbehaviour: ProtoAny,
    /// signer address
    pub signer: String,
}

pub struct Misbehaviour {
    pub client_id: ClientId,
    pub header1: Header,
    pub header2: Header,
}
```

Message: `ConnectionOpenInit`

Messages `ConnectionOpenInit`, `ConnectionOpenTry`, `ConnectionOpenAck`, and `ConnectionOpenConfirm` form a handshake for establishing a bidirectional ICS-003 connection between two chains (henceforth A and B), such that each chain represents the other as a client and that each has knowledge of the corresponding client and connection IDs.

The `ConnectionOpenInit` assigns the next available connection ID on A for the (A, B) connection pair as the part of the path in A's state to store the `ConnectionEnd` (with `State::Init`) for the subsequent messages to reference in proofs.

```
pub struct MsgConnectionOpenInit {
    /// ClientId on chain A that the connection is being opened for
    pub client_id_on_a: ClientId,
    pub counterparty: Counterparty,
    pub version: Option<Version>,
    pub delay_period: Duration,
    pub signer: String,
```

```
}

pub struct ConnectionEnd {
    pub state: State,
    pub client_id: ClientId,
    pub counterparty: Counterparty,
    pub versions: Vec<Version>,
    pub delay_period: Duration,
}

pub enum State {
    Uninitialized = 0isize,
    Init = 1isize,
    TryOpen = 2isize,
    Open = 3isize,
}

pub struct Counterparty {
    pub client_id: ClientId,
    pub connection_id: Option<ConnectionId>,
    pub prefix: MerklePrefix,
}
```

Message: `ConnectionOpenTry`

The `ConnectionOpenTry` message has B verify that the following are committed to in A's state:

- The `ConnectionEnd` constructed from the data in the `ConnectionOpenTry` message (with `State::Init`) is at the corresponding path based on A's connection ID for (A, B).
- B's client state is committed to by A at the corresponding path for B's client ID on A.
- B's consensus state is committed to by A at the corresponding path for B's client ID on A.

B allocates its own connection ID for (A, B) and stores the corresponding `ChannelEnd` (with `State::TryOpen`) in its state under that ID, with a compatible version selected from A's declared supported versions.

```
pub struct MsgConnectionOpenTry {
    /// ClientId on B that the connection is being opened for
    pub client_id_on_b: ClientId,
    /// ClientState of client tracking chain B on chain A
    pub client_state_of_b_on_a: Any,
    /// ClientId, ConnectionId and prefix of chain A
    pub counterparty: Counterparty,
    /// Versions supported by chain A
    pub versions_on_a: Vec<Version>,
```

```
    /// proof of ConnectionEnd stored on Chain A during ConnOpenInit
    pub proof_conn_end_on_a: MerkleProof,
    /// proof that chain A has stored ClientState of chain B on its client
    pub proof_client_state_of_b_on_a: MerkleProof,
    /// proof that chain A has stored ConsensusState of chain B on its client
    pub proof_consensus_state_of_b_on_a: MerkleProof,
    /// Height at which all proofs in this message were taken
    pub proofs_height_on_a: Height,
    /// height of latest header of chain A that updated the client on chain B
    pub consensus_height_of_b_on_a: Height,
    pub delay_period: Duration,
    pub signer: String,
    pub proof_consensus_state_of_b: Option<MerkleProof>,

    #[deprecated(since = "0.22.0")]
    /// Only kept here for proper conversion to/from the raw type
    pub previous_connection_id: String,
}
```

Message: `ConnectionOpenAck`

The `ConnectionOpenAck` message has A check that it has a `ConnectionEnd` with B in `State::Init` and that the version B chose is one of its supported versions.

A then verifies that the following are committed to in B's state:

- The `ConnectionEnd` constructed from the data in the `ConnectionOpenTry` message (with `State::TryOpen`)is at the corresponding path based on B's connection ID for (A, B).
- A's client state is committed to by B at the corresponding path for A's client ID on B.
- A's consensus state is committed to by B at the corresponding path for A's client ID on B.

A then updates its `ConnectionEnd` to `State::Open` or `State::TryOpen` (see Finding 3.4. ↗), with the agreed-on version and with B's connection ID for (A, B) that it committed to (replacing the one provided in `ConnectionOpenInit`'s counterparty).

```
pub struct MsgConnectionOpenAck {
    /// ConnectionId that chain A has chosen for it's ConnectionEnd
    pub conn_id_on_a: ConnectionId,
    /// ConnectionId that chain B has chosen for it's ConnectionEnd
    pub conn_id_on_b: ConnectionId,
    /// ClientState of client tracking chain A on chain B
    pub client_state_of_a_on_b: Any,
    /// proof of ConnectionEnd stored on Chain B during ConnOpenTry
    pub proof_conn_end_on_b: MerkleProof,
    /// proof of ClientState tracking chain A on chain B
```

```
    pub proof_client_state_of_a_on_b: MerkleProof,
    /// proof that chain B has stored ConsensusState of chain A on its client
    pub proof_consensus_state_of_a_on_b: MerkleProof,
    /// Height at which all proofs in this message were taken
    pub proofs_height_on_b: Height,
    /// height of latest header of chain A that updated the client on chain B
    pub consensus_height_of_a_on_b: Height,
    /// optional proof of the consensus state of the host chain, see:
    <https://github.com/cosmos/ibc/pull/839>
    host_consensus_state_proof: Option<MerkleProof>,
    pub version: Version,
    pub signer: String,
}
```

Message: `ConnectionOpenConfirm`

The `ConnectionOpenConfirm` message has B check that its `ConnectionEnd` is in `State::TryOpen`, then verifies that A has committed to a `ConnectionEnd` in `State::Open` with the corresponding data in the `ConnectionOpenConfirm` message, and then updates its `ConnectionEnd` to `State::Open`. This concludes the handshake.

```
pub struct MsgConnectionOpenConfirm {
    /// ConnectionId that chain B has chosen for it's ConnectionEnd
    pub conn_id_on_b: ConnectionId,
    /// proof of ConnectionEnd stored on Chain A during ConnOpenInit
    pub proof_conn_end_on_a: MerkleProof,
    /// Height at which `proof_conn_end_on_a` in this message was taken
    pub proof_height_on_a: Height,
    pub signer: String,
}
```

Message: `ChannelOpenInit`

Messages `ChannelOpenInit`, `ChannelOpenTry`, `ChannelOpenAck`, and `ChannelOpenConfirm` form a handshake for establishing bidirectional ICS-004 data channels between chains that are transitively connected by ICS-003 connections. Currently, `penumbra-ibc` only supports channels with exactly one connection. In other words, for a channel to be established between A and B, there must be a direct connection between A and B rather than connections between (A, C) and (C, B). The `AppHandler` trait additionally allows application-specific checks to be added to each step of the handshake if the port ID is specified as `"transfer"` (e.g., penumbra-shielded-pool and astria-sequencer's ICS-020 implementations enforce that their channels are unordered through these callbacks).

ChannelOpenInit has A check

- that the connection to be established is exactly one hop
- that the specified (`channel_id`, `port_id_on_a`) is unused in A's state with the next available `channel_id` (this check ensures `ChannelOpenInit` is idempotent/immune to replay attacks)
- that an (A, B) connection exists in A's state (but not necessarily in `State::Open`, to reduce the number of round trips when establishing a connection and channel on that connection concurrently).

A then generates the next sequential channel ID, stores a `ChannelEnd` in `State::Init` in its state under (`channel_id`, `port_id_on_a`), and initializes its `{send, recv, ack}` sequence numbers to 1.

```
pub struct MsgChannelOpenInit {
    pub port_id_on_a: PortId,
    pub connection_hops_on_a: Vec<ConnectionId>,
    pub port_id_on_b: PortId,
    pub ordering: Order,
    pub signer: String,
    /// Allow a relayer to specify a particular version by providing a
    non-empty version string
    pub version_proposal: Version,
}

pub struct ChannelEnd {
    pub state: State,
    pub ordering: Order,
    pub remote: Counterparty,
    pub connection_hops: Vec<ConnectionId>,
    pub version: Version,
}

pub struct Counterparty {
    pub port_id: PortId,
    pub channel_id: Option<ChannelId>,
}

pub enum Order {
    None = 0isize,
    Unordered = 1isize,
    Ordered = 2isize,
}

pub enum State {
    Uninitialized = 0isize,
    Init = 1isize,
    TryOpen = 2isize,
    Open = 3isize,
```

```
    Closed = 4isize,
}
```

Message: `ChannelOpenTry`

The `ChannelOpenTry` message has B check

- that the connection to be established is exactly one hop
- that A has committed a `ChannelEnd` in `State::Init` with the provided ordering and a single connection to B

B does not check that the `port_id_on_b` specified by A is free in B's state, which is fine since the `ChannelEnds` are stored under a (`ChannelId`, `PortId`) pair, and the `ChannelId` is fresh.

B then generates its next free channel ID, creates a `ChannelEnd` in `State::TryOpen`, stores it in its state under the specified port ID, and initializes its `{send, recv, ack}` sequence numbers to 1.

```
pub struct MsgChannelOpenTry {
    pub port_id_on_b: PortId,
    pub connection_hops_on_b: Vec<ConnectionId>,
    pub port_id_on_a: PortId,
    pub chan_id_on_a: ChannelId,
    pub version_supported_on_a: Version,
    pub proof_chan_end_on_a: MerkleProof,
    pub proof_height_on_a: Height,
    pub ordering: Order,
    pub signer: String,

    #[deprecated(since = "0.22.0")]
    /// Only kept here for proper conversion to/from the raw type
    pub previous_channel_id: String,
    #[deprecated(since = "0.22.0")]
    /// Only kept here for proper conversion to/from the raw type
    pub version_proposal: Version,
}
```

Message: `ChannelOpenAck`

The `ChannelOpenAck` message has A check

- that its `ChannelEnd` is in `State::Init` or `State::TryOpen` (see Finding 3.4. ↗)
- that its (A, B) `ConnectionEnd` is in `State::Open`
- that B has committed to a `ChannelEnd` in `State::TryOpen` with data consistent with the expected new state

A then sets its `ChannelEnd` state to `State::Open` and updates the channel ID/version to match B's state.

```
pub struct MsgChannelOpenAck {
    pub port_id_on_a: PortId,
    pub chan_id_on_a: ChannelId,
    pub chan_id_on_b: ChannelId,
    pub version_on_b: Version,
    pub proof_chan_end_on_b: MerkleProof,
    pub proof_height_on_b: Height,
    pub signer: String,
}
```

Message: `ChannelOpenConfirm`

The `ChannelOpenConfirm` message has B check

- that its `ChannelEnd` is in `State::TryOpen`
- that its (A, B) `ConnectionEnd` is in `State::Open`
- that A has committed to a `ChannelEnd` in `State::Open` with data consistent with B's state

B then sets its `ChannelEnd` state to `State::Open`. This concludes the handshake.

```
pub struct MsgChannelOpenConfirm {
    pub port_id_on_b: PortId,
    pub chan_id_on_b: ChannelId,
    pub proof_chan_end_on_a: MerkleProof,
    pub proof_height_on_a: Height,
    pub signer: String,
}
```

Message: `ChannelCloseInit`

Both `ChannelCloseInit` and `ChannelCloseConfirm` form a handshake for closing an existing open channel.

For `ChannelCloseInit`, A checks that the channel is not already in `State::Closed` and that the (A, B) connection is in `State::Open`, then sets the channel to `State::Closed`.

```
pub struct MsgChannelCloseInit {
    pub port_id_on_a: PortId,
    pub chan_id_on_a: ChannelId,
    pub signer: String,
```

```
}
```

Message: `ChannelCloseConfirm`

For `ChannelCloseConfirm`, B checks that the channel is not already in `State::Closed` and that the (A, B) connection is in `State::Open`, and it verifies that A's state contains the channel in a form consistent with B's state but with the state changed to `State::Closed`, and then sets its own channel to `State::Closed`.

```
pub struct MsgChannelCloseConfirm {
    pub port_id_on_b: PortId,
    pub chan_id_on_b: ChannelId,
    pub proof_chan_end_on_a: MerkleProof,
    pub proof_height_on_a: Height,
    pub signer: String,
}
```

Message: `RecvPacket`

The `RecvPacket` message delivers a packet along an existing (A, B) channel. Without loss of generality, messages are said to be sent from A to B, but since the channel is bidirectional, these are not the same as the A and B in channel establishment.

When receiving a packet, B checks

- that the (A, B) channel is in `State::Open`
- that the packet's port and channel IDs match the channel's sender's port and channel IDs
- that the (A, B) connection is in `State::Open`
- that the packet has not timed out
- that the packet was committed to in A's state
- if the channel is ordered, that its sequence number matches B's recv sequence number
- if the channel is unordered, that it has not already been processed

B then increments its recv sequence number (for ordered channels) or marks the packet as processed (for unordered channels).

```
pub struct MsgRecvPacket {
    /// The packet to be received
    pub packet: Packet,
    /// Proof of packet commitment on the sending chain
    pub proof_commitment_on_a: MerkleProof,
    /// Height at which the commitment proof in this message were taken
    pub proof_height_on_a: Height,
```

```
    /// The signer of the message
    pub signer: String,
}

pub struct Packet {
    pub sequence: Sequence,
    pub port_on_a: PortId,
    pub chan_on_a: ChannelId,
    pub port_on_b: PortId,
    pub chan_on_b: ChannelId,
    pub data: Vec<u8>,
    pub timeout_height_on_b: TimeoutHeight,
    pub timeout_timestamp_on_b: Timestamp,
}
```

Message: `Acknowledgement`

An `Acknowledgement` message tells A that B received a packet from A at a particular time.

When receiving an acknowledgment, A checks

- that the (A, B) channel is in `State::Open`
- that the acknowledged packet's port and channel IDs match the channel's receiver's IDs
- that the (A, B) connection is in `State::Open`
- that A committed to the claimed received packet
- that the send of the acknowledgment of the packet is in B's state
- if the channel is ordered, that its sequence number matches A's ack sequence number

If the channel is ordered, A then increments its ack sequence number.

A then deletes the packet from its set of pending packets it has sent but not acknowledged.

```
pub struct MsgAcknowledgement {
    pub packet: Packet,
    pub acknowledgement: Vec<u8>,
    /// Proof of packet acknowledgement on the receiving chain
    pub proof_acked_on_b: MerkleProof,
    /// Height at which the commitment proof in this message were taken
    pub proof_height_on_b: Height,
    pub signer: String,
}
```

Message: `Timeout`

A `Timeout` message tells A that a packet it sent to B has not been received by the packet's expiration time.

When receiving a timeout, A checks

- that the (A, B) channel is in `State::Open`
- that the packet's port and channel IDs match the channel's receiver's IDs
- that the (A, B) connection exists
- that B's latest state has a later time than the packet's expiry
- that A committed to the claimed packet
- if the channel is ordered, that B's recv sequence number does not match the packet (see Finding 3.11. ↗)
- if the channel is unordered, that the packet receipt is absent from B's state

A then deletes the packet from its set of pending packets it has sent but not acknowledged, and if the channel is ordered, closes it.

```
pub struct MsgTimeout {
    pub packet: Packet,
    pub next_seq_recv_on_b: Sequence,
    pub proof_unreceived_on_b: MerkleProof,
    pub proof_height_on_b: Height,
    pub signer: String,
}
```

Message: `Unknown`

Messages that are not one of the above types are encapsulated into the `Unknown` variant, and such messages are rejected.

```
pub enum IbcRelay {
    // ...
    Unknown(pbjson_types::Any),
}
```

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the mainnet.

During our assessment on the scoped Penumbra IBC crates, we discovered 11 findings. No critical issues were found. Two findings were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

DRAFT