Zellic

April 22, 2024

Prepared for
Jordan Oroshiba
Astria

Prepared by
Syed Faraz Abrar
William Bowling
Avraham Weinstock
Zellic

Astria Shared Sequencer
Blockchain Security Assessment

Contents

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 ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Astria from March 4th to April 17th, 2024. During this engagement, Zellic reviewed Astria Shared Sequencer'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:

- Could an attacker cause a loss of funds for users?
- Can the rollup trust the data coming from the conductor?
- Can each node operator receive their full blocks?
- Are the blocks executed correctly and in order?

1.3. Non-goals and Limitations

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

- 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.

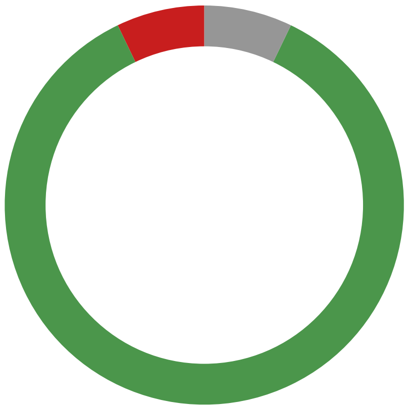
1.4. Results

During our assessment on the scoped Astria Shared Sequencer Crates, we discovered 14 findings. One critical issue was found, 12 were low impact, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	0
🟨 Medium	0
🟩 Low	12
⬜ Informational	1



2. Introduction

2.1. About Astria Shared Sequencer

Astria contributed the following description of Astria Shared Sequencer:

> Astria is a Shared Sequencer Network with Celestia underneath. It provides ordering guarantees with soft commitments to chains, relying on Celestia's DAS network to provide firm commitments and broad data availability.

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 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.

2.3. Scope

The engagement involved a review of the following targets:

Astria Shared Sequencer Crates

Repository	https://github.com/astriaorg/astria ↗
Version	astria: `105474286ea1803244dcf2851bb5a84f5e16daa8`
Programs	• astria-conductor/src/* • astria-sequencer/src/* • astria-merkle/src/* • astria-sequencer-relayer/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 and two calendar days.

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	Kick-off call
March 4, 2024	Start of primary review period
April 17, 2024	End of primary review period

3. Detailed Findings

3.1. Prepare proposal must obey the max bytes limit

Target	astria-sequencer/src/app.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When a `PrepareProposal` request is sent by CometBFT, there is a `max_tx_bytes` parameter that indicates that maximum size of all the transactions that can be returned. See the spec for more details at https://docs.cometbft.com/v0.37/spec/abci/abci++_methods#parameters-and-types ↗.

The implementation of `prepare_proposal` currently does not check the size of the transactions that are being returned against the `max_tx_bytes` limit:

```
#[instrument(name = "App::prepare_proposal", skip_all)]
pub(crate) async fn prepare_proposal(
    &mut self,
    prepare_proposal: abci::request::PrepareProposal,
    storage: Storage,
) -> anyhow::Result<abci::response::PrepareProposal> {
    self.is_proposer = true;
    self.update_state_for_new_round(&storage);

    let (signed_txs, txs_to_include)
    = self.execute_block_data(prepare_proposal.txs).await;

    let deposits = self
        .state
        .get_block_deposits()
        .await
        .context("failed to get block deposits in prepare_proposal")?;

    // generate commitment to sequence::Actions and deposits and commitment to
    the rollup IDs
    // included in the block
    let res = generate_rollup_datas_commitment(&signed_txs, deposits);

    Ok(abci::response::PrepareProposal {
        txs: res.into_transactions(txs_to_include),
    })
}
```

Impact

If the list of transactions is over the `max_tx_bytes` limit, then CometBFT will immediately panic, leading to a chain halt.

For reference, this is the flow in CometBFT:

1. CometBFT sends the `PrepareProposal` request to the Astria Sequencer here ↗.
2. The transactions returned are validated here ↗.
3. Validation will return an error if the `max_tx_bytes` is exceeded here ↗.
4. When `CreateProposalBlock()` returns the error, the code will panic immediately here ↗.

Recommendations

The returned list of transactions should be checked against the `max_tx_bytes` limit to ensure that it is below the limit.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `473041a8` ↗.

3.2. Incorrect `is_source` logic

Target	astria-sequencer/src/ibc/ics20_{transfer,withdrawal}.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When transferring or receiving assets via IBC, a check needs to be performed in order to determine whether the asset originated from Astria or if it was from an external chain. From the ics-020-fungible-token-transfer spec ↗, the logic is as follows:

```
prefix = "{sourcePort}/{sourceChannel}/"
// we are the source if the denomination is not prefixed
source = denomination.slice(0, len(prefix)) !== prefix
if source {
```

The two implementations in Astria are as follows:

```
// ics20_withdrawal.rs
fn is_source(source_port: &PortId, source_channel: &ChannelId, asset: &Denom)
    -> bool {
    let prefix = format!("{source_port}/{source_channel}/");
    !asset.prefix_is(&prefix)
}

// ics20_transfer.rs
fn is_source(
    source_port: &PortId,
    source_channel: &ChannelId,
    asset: &Denom,
    is_refund: bool,
) -> bool {
    let prefix = format!("{source_port}/{source_channel}/");
    if is_refund {
        !asset.prefix_is(&prefix)
    } else {
        asset.prefix_is(&prefix)
    }
}
```

This implementation seems correct, but due to the way the `Denom` is created, the prefix will never end in a slash:

```rust
impl Denom {
    pub fn prefix_is(&self, prefix: &str) -> bool {
        self.prefix == prefix
    }
}

impl From<String> for Denom {
    fn from(denom: String) -> Self {
        let Some((prefix, base_denom)) = denom.rsplit_once('/') else {
            return Self {
                id: Id::from_denom(&denom),
                base_denom: denom,
                prefix: String::new(),
            };
        };

        Self {
            id: Id::from_denom(&denom),
            base_denom: base_denom.to_string(),
            prefix: prefix.to_string(),
        }
    }
}
```

For example, when parsing the denom `transfer/channel-0/stake`, the prefix will be `transfer/channel-0` and the `base_denom` will be `stake`, causing the prefix in `is_source` to never match.

Impact

When performing an `Ics20Withdrawal`, the `is_source` function will always return true, causing the funds to always be put into the escrow account.

When receiving funds via an `ics20` transfer, `is_source` will always return false, causing the incoming denom to always have an additional prefix prepended. This makes it impossible to send funds back to the original chain.

Recommendations

The `prefix` in the `is_source` functions should have the slash removed from the end, or the `Denom` prefix should always include the slash at the end.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `2c0d2b11` ↗.

3.3. The prefix of returned assets is not removed

Target	astria-sequencer/src/ibc/ics20_transfer.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When an asset that originated from Astria is returned via an IBC transfer, the escrow balance for that channel is checked to ensure that the original asset was transferred:

```
if is_source(source_port, source_channel, &denom, is_refund) {
    // sender of packet (us) was the source chain
    // subtract balance from escrow account and transfer to user

    let escrow_balance = state
        .get_ibc_channel_balance(source_channel, denom.id())
        .await
        .context("failed to get IBC channel balance in
    execute_ics20_transfer")?;

    let user_balance = state.get_account_balance(recipient,
    denom.id()).await?;
    state
        .put_ibc_channel_balance(
            source_channel,
            denom.id(),
            escrow_balance
                .checked_sub(packet_amount)
                .ok_or(anyhow::anyhow!(
                    "insufficient balance in escrow account to transfer tokens"
                ))?,
        )
        .context("failed to update escrow account balance in
    execute_ics20_transfer")?;
```

The issue is that the incoming assets' denom will be prefixed with `{source_port}/{source_channel}/`, which is never removed. As the escrow balance was updated with the unprefixed denom, the balance will be zero and the transfer will fail.

Impact

When someone tries to return assets that originated from Astria, the transfer will fail due to the escrow balance being zero, meaning once assets are transferred out, they cannot be returned.

Recommendations

The prefix should be removed from the denom before checking the escrow balance.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `b58b65e8` ↗.

3.4. Incorrect channel when checking escrow balance

Target	astria-sequencer/src/ibc/ics20_transfer.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When an asset that originated from Astria is returned via an IBC transfer, the escrow balance for that channel is checked to ensure that the original asset was transferred:

```
if is_source(source_port, source_channel, &denom, is_refund) {
    // sender of packet (us) was the source chain
    // subtract balance from escrow account and transfer to user

    let escrow_balance = state
        .get_ibc_channel_balance(source_channel, denom.id())
        .await
        .context("failed to get IBC channel balance in
    execute_ics20_transfer")?;

    let user_balance = state.get_account_balance(recipient,
    denom.id()).await?;
    state
        .put_ibc_channel_balance(
            source_channel,
            denom.id(),
            escrow_balance
                .checked_sub(packet_amount)
                .ok_or(anyhow::anyhow!(
                    "insufficient balance in escrow account to transfer tokens"
                ))?,
        )
        .context("failed to update escrow account balance in
    execute_ics20_transfer")?;
```

The issue is that the escrow balance is checked using the `source_channel` instead of the `dest_channel`, as the spec stipulates ↗.

Impact

This could either cause the escrow balance to be zero or the balance for the wrong channel to be checked, allowing the escrow balance of the wrong channel to be updated.

Recommendations

The escrow balance should be checked using the `dest_channel` instead of the `source_channel`.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `b58b65e8` ↗.

3.5. Out-of-bounds access fetching block by hash

Target	astria-sequencer/src/api_state_ext.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When querying a sequencer block, the rollup transactions are enumerated and added to a vector:

```
let mut rollup_transactions = Vec::with_capacity(rollup_ids.len());
for (i, id) in rollup_ids.iter().enumerate() {
    let key = rollup_data_by_hash_and_rollup_id_key(hash, id);
    let raw = self
        .get_raw(&key)
        .await
        .context("failed to read rollup data by block hash and rollup ID from
    state")?;
    if let Some(raw) = raw {
        let raw = raw.as_slice();
        let rollup_data = raw::RollupTransactions::decode(raw)
            .context("failed to decode rollup data from raw bytes")?;
        rollup_transactions[i] = rollup_data;
    }
}
```

The issue is that while the vector has the right capacity, it still has a length of zero and so assigning to index `i` will cause a panic.

Impact

The `get_sequencer_block_by_hash` function is used by the `GetSequencerBlock` RPC call, so anything that tries to use this RPC call will recieve an error such as the `BlockStream` required by the relayer.

Recommendations

Each element should be pushed to the vector instead of assigning to an index; since it has been preallocated with the correct capacity, it should not need to be resized.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `3516a7e3` ↗.

3.6. Duplicate rollup ids

Target	astria-sequencer/src/src/bridge/state_ext.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When fetching the list of rollup IDs for the current block, the `DEPOSIT_PREFIX` stream is iterated over and any rollup IDs found are added to a vector.

```
async fn get_deposit_rollup_ids(&self) -> Result<Vec<RollupId>> {
    let mut stream =
    std::pin::pin!(self.nonverifiable_prefix_raw(DEPOSIT_PREFIX.as_bytes()));
    let mut rollup_ids = Vec::new();
    while let Some(Ok((key, _))) = stream.next().await {
        // the deposit key is of the form "deposit/{rollup_id}/{nonce}"
        let key_str =
            String::from_utf8(key).context("failed to convert deposit key to
    string")?;
        let key_parts = key_str.split('/').collect::<Vec<_>>();
        if key_parts.len() != 3 {
            continue;
        }
        let rollup_id_bytes =
            hex::decode(key_parts[1]).context("invalid rollup ID hex string")?;
        let rollup_id =
            RollupId::try_from_slice(&rollup_id_bytes).context("invalid rollup
    ID bytes")?;
        rollup_ids.push(rollup_id);
    }
    Ok(rollup_ids)
}
```

The issue is that if there are multiple deposits for the same rollup, then the returned vector will contain duplicate rollup ids.

Impact

Currently the two places that use `deposit_rollup_ids` are `get_block_deposits` and `clear_block_deposits`, both of which only cause unneeded computations to be made as the first is putting the rollup IDs and the deposits into a hash map and the second is clearing them from the state.

Recommendations

The rollup IDs should be added to a set instead of a vector to ensure that there are no duplicates, or documentation should be added to ensure that anyone using the function is aware that there may be duplicates.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `f9b3bb45` ↗.

3.7. Duplicate transaction inconsistency

Target	astria-sequencer/src/app.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When a `PrepareProposal` request is made to the current proposer, it is possible that the request may contain duplicate transactions. This is explicitly mentioned in the spec at https://docs.cometbft.com/v0.37/spec/abci/abci++_methods ↗:

> CometBFT does NOT provide any additional validity checks (such as checking for duplicate transactions).

Normally, this would not be an issue, as even if the duplicate transaction was included, the second one would fail as the nonce would be incorrect.

The issue is that when the proposer executes the second transaction in `execute_block_data`, the results are stored in the `execution_result` cache by the transaction hash, replacing the results of the first transaction.

```
// store transaction execution result, indexed by tx hash
match self.deliver_tx(signed_tx.clone()).await {
    Ok(events) => {
        self.execution_result.insert(tx_hash.into(), Ok(events));
        signed_txs.push(signed_tx);
        validated_txs.push(tx);
        block_sequence_data_bytes += tx_sequence_data_bytes;
    }
    Err(e) => {
        debug!(
            transaction_hash = %telemetry::display::hex(&tx_hash),
            error = AsRef::<dyn std::error::Error>::as_ref(&e),
            "failed to execute transaction, not including in block"
        );
        excluded_tx_count += 1;
        self.execution_result.insert(tx_hash.into(), Err(e));
    }
}
```

This will cause the second transaction to be removed from the list of transactions to be included, but the first transaction will still be included. When the other validators process the proposal, they will only see one transaction, which will succeed, causing them to have a different state than the proposer.

Impact

If there are three or fewer validators, then this will cause a consensus failure as the proposal requires more than 1/3 of the validators to agree. If there are more than three, then a new proposer will be chosen and the process will repeat until the duplicate transaction is dropped from the mempool.

Recommendations

If a transaction has already been executed and is in the cache, the proposer should not execute it a second time.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `dfebc587` ↗.

3.8. Transfer of zero amounts is allowed

Target	astria-sequencer/src/accounts/state_ext.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When performing a transfer between Astria accounts, there is no check on the `TransferAction` to ensure that the `amount` being transferred is not zero, only that the sending user has a balance greater to or equal to `amount` for the requested asset.

This would not normally be an issue, but in this case it causes an issue with the `get_account_balances` function as it loops over all of the stored balances (even if they are zero) and tries to fetch the `get_ibc_asset` for each asset ID found:

```
async fn get_account_balances(&self, address: Address) ->
    Result<Vec<AssetBalance>> {
        use crate::asset::state_ext::StateReadExt as _;

        let prefix = format!("{}/balance/",
    storage_key(&address.encode_hex::<String>()));
        let mut balances: Vec<AssetBalance> = Vec::new();

        let mut stream = std::pin::pin!(self.prefix_keys(&prefix));
        while let Some(Ok(key)) = stream.next().await {
            // ... [snip]
            let asset_id_str = key
                .strip_prefix(&prefix)
                .context("failed to strip prefix from account balance key")?;
            let asset_id_bytes = hex::decode(asset_id_str).context("invalid
    asset id bytes")?;

            let asset_id = asset::Id::try_from_slice(&asset_id_bytes)
                .context("failed to parse asset id from account balance key")?;
            let Balance(balance) =
                Balance::try_from_slice(&value).context("invalid balance
    bytes")?;

             // ... [snip]
            let denom = self.get_ibc_asset(asset_id).await?;
            balances.push(AssetBalance {
                denom,
                balance,
            });
        }
        Ok(balances)
    }
```

If a made-up asset denom was used, then the check to `get_ibc_asset` will fail, as the asset does not exist, causing the user to be unable to check their balance.

Impact

A malicious user can send a zero transfer of a nonexistent asset to another user, causing them to be unable to check their balance.

Recommendations

A `check_stateless` check could be added to ensure that the `amount` being transferred is greater than zero.

Remediation

This issue has been acknowledged by Astria.

3.9. Proposal time-out leads to differing chain states

Target	astria-sequencer/src/accounts/action.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When a `prepare_proposal` is called for the current proposer, the `is_proposer` flag is set to true. This is then checked when processing the proposal so that the execution can be skipped:

```
pub(crate) async fn prepare_proposal(
    &mut self,
    prepare_proposal: abci::request::PrepareProposal,
    storage: Storage,
) -> anyhow::Result<abci::response::PrepareProposal> {
    self.is_proposer = true;
    self.update_state_for_new_round(&storage);

// [snip]

pub(crate) async fn process_proposal(
    &mut self,
    process_proposal: abci::request::ProcessProposal,
    storage: Storage,
) -> anyhow::Result<()> {
    // if we proposed this block (ie. prepare_proposal was called directly
    before this), then
    // we skip execution for this `process_proposal` call.
    //
    // if we didn't propose this block, `self.is_proposer` will be `false`, so
    // we will execute the block as normal.
    if self.is_proposer {
        debug!("skipping process_proposal as we are the proposer for this
    block");
        self.is_proposer = false;
        self.executed_proposal_hash = process_proposal.hash;
        return Ok(());
    }
```

The issue is that it is possible for the proposer to time out, causing a round to begin and a new proposer to be chosen. This will leave the old proposer with the `is_proposer` flag set to true, even though they are not the current proposer. If the new proposer produces a different block, then the old proposer will reset the state changes and execute the transactions in `deliver_tx`, but the commitments will not be checked:

```
// If we previously executed txs in a different proposal than is being
    processed reset
// cached state changes.
if self.executed_proposal_hash != begin_block.hash {
    self.update_state_for_new_round(&storage);
}

// [snip]

pub(crate) async fn deliver_tx_after_proposal(
    &mut self,
    tx: abci::request::DeliverTx,
) -> Option<anyhow::Result<Vec<abci::Event>>> {
    self.current_sequencer_block_builder
        .as_mut()
        .expect(
            "begin_block must be called before deliver_tx, thus \
                current_sequencer_block_builder must be set",
        )
        .push_transaction(tx.tx.to_vec());

    if self.processed_txs < 2 {
        self.processed_txs += 1;
        return Some(Ok(vec![]));
    }

    // When the hash is not empty, we have already executed and cached the
    results
    if !self.executed_proposal_hash.is_empty() {
        let tx_hash: [u8; 32] = sha2::Sha256::digest(&tx.tx).into();
        return self.execution_result.remove(&tx_hash);
    }
```

Impact

If a validator times out when proposing a block, they will still think that they are the proposer when `process_proposal` is called and automatically vote for the block. If a malicious proposer is able to create a batch of transactions that causes block proposals to be slow (for instance using 3.12. ↗), then many validators could be put into this state, causing either a consensus failure or arbitrary commitments to be supplied.

Recommendations

If the `is_proposer` is set to true when `process_proposal` is called, the supplied `proposer_address` in the request should also be checked to ensure that it has not changed.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `f26440a0` ↗.

3.10. Mint action can overflow the balance

Target	astria-sequencer/src/mint/action.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

If the `mint` feature is enabled, it allows the `sudo` address to mint arbitrary amounts of the native asset to an account:

```rust
async fn execute<S: AccountStateWriteExt + AccountStateReadExt>(
    &self,
    state: &mut S,
    _: Address,
) -> Result<()> {
    let native_asset = get_native_asset().id();

    let to_balance = state
        .get_account_balance(self.to, native_asset)
        .await
        .context("failed getting `to` account balance")?;
    state
        .put_account_balance(self.to, native_asset, to_balance + self.amount)
        .context("failed updating `to` account balance")?;
    Ok(())
}
```

The issue is that `to_balance + self.amount` could overflow, resetting the balance back to zero.

Impact

If the `mint` feature is enabled, the sudo account can mint a large amount of the native asset to an account, causing the balance to overflow and reset to zero.

Recommendations

The new `increase_balance` helper function should be used to ensure that the balance does not overflow.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `0c54b993` ↗.

3.11. Only some IBC requests can fail

Target	astria-sequencer/src/service/consensus.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When handling IBC requests, the ABCI specification ↗ mentions that the requests `Query`, `CheckTx`, and `DeliverTx` can fail and return an error code. An error in any of the other requests is considered a critical issue, and the application must crash as `CometBFT` has no correct way to handle an error.

Currently, the `handle_request` function will return an error for any request that fails, which is then dropped if it fails to send.

Impact

If an error occurs in a request that is not supposed to fail, the application could be in an inconsistent state and cause `CometBFT` to have a consensus failure.

Recommendations

The `handle_request` handler should use `expect` on the requests that are not supposed to fail, causing the application to crash if an error occurs.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `f78f6d85` ↗.

3.12. List of assets in bridge init is unbounded

Target	astria-sequencer/src/bridge/init_bridge_account_action.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When creating a new Bridge account, the `InitBridgeAccountAction` is used, which takes a list of asset IDs that the bridge will accept:

```
pub struct InitBridgeAccountAction {
    // the rollup ID to register for the sender of this action
    pub rollup_id: RollupId,
    // the assets accepted by the bridge account
    pub asset_ids: Vec<asset::Id>,
    // the fee asset which to pay this action's fees with
    pub fee_asset_id: asset::Id,
}
```

The issue is that the only check on the `asset_ids` is that the length is greater than zero; there is no check on the length of the list.

Impact

A malicious actor could create a bridge account with a large amount of asset IDs, while still only paying the standard `INIT_BRIDGE_ACCOUNT_FEE` fee of 48. When a `BridgeLockAction` action is performed, the large list needs to be fetched from storage and iterated through to see if the asset ID is in the list. Since this happens in a `check_stateful` handler, the action will pass the `CheckTx` handler and make it into the mempool.

Recommendations

The current check for `!self.asset_ids.is_empty()` should be moved into the `check_stateless` handler, and a maximum length should be enforced.

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `89afc238` ↗.

3.13. Duplicate Celestia rollup blob causes panic

Target	astria-conductor/src/celestia/mod.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When the conductor receives a new sequencer blob from Celestia, it verifies the hash against the sequencer and then fetches the relevant rollup blobs:

```
async fn process_sequencer_blob(
    client: HttpClient,
    verifier: BlockVerifier,
    celestia_height: u64,
    rollup_namespace: Namespace,
    sequencer_blob: CelestiaSequencerBlob,
) -> eyre::Result<ReconstructedBlock> {
    verifier
        .verify_blob(&sequencer_blob)
        .await
        .wrap_err("failed validating sequencer blob retrieved from celestia")?;
    let mut rollup_blobs = client
        .get_rollup_blobs_matching_sequencer_blob(
            celestia_height,
            rollup_namespace,
            &sequencer_blob,
        )
        .await
        .wrap_err("failed fetching rollup blobs from celestia")?;
    debug!(
        %celestia_height,
        number_of_blobs = rollup_blobs.len(),
        "received rollup blobs from Celestia"
    );
    ensure!(
        rollup_blobs.len() <= 1,
        "received more than one celestia rollup blob for the given namespace
    and height"
    );
```

The issue is that Celestia has no restrictions on duplicate blobs, so it is possible for a duplicate blob to be stored for the rollup at the specified height. This would cause the `ensure` block checking the length of `rollup_blobs` to panic, crashing the conductor.

Impact

A malicious actor could send a duplicate valid rollup blob to Celestia, causing the conductor to panic and crash.

Recommendations

The conductor should either return the first blob for the given namespace and height or explicitly check for duplicates. Alternatively, the check could be removed and let the canonical chain handle the duplicate blobs.

Remediation

We have reviewed the described changes and examined the code in pull request 946 ↗. The proposed modifications should resolve the original issue. However, due to the substantial amount of new and refactored code, we are unable to fully review the entire pull request within the allocated remediation time.

3.14. TXs that are too large are rejected without error

Target	astria-composer/src/searcher/executor/bundle_factory/mod.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The Composer component's Collector will collect transactions from rollup nodes and send them to the Executor, which will attempt to construct bundles of transactions to be submitted to the Sequencer.

In this case, if one of the rollup transactions is too large, the Executor simply drops the transaction and continues on as if nothing happened.

```rust
pub(super) async fn run_until_stopped(mut self) -> eyre::Result<()> {
    // [ ... ]

    // receive new seq_action and bundle it
    Some(seq_action) = self.serialized_rollup_transactions_rx.recv() => {
        let rollup_id = seq_action.rollup_id;
        if let Err(e) = bundle_factory.try_push(seq_action) {
                warn!(
                    rollup_id = %rollup_id,
                    error = &e as &StdError,
                    "failed to bundle sequence action: too large. sequence
    action is dropped."
                );
        }
    }
    // [ ... ]
}

/// Buffer `seq_action` into the current bundle. If the bundle won't fit
    `seq_action`, flush
/// `curr_bundle` into the `finished` queue and start a new bundle
pub(super) fn try_push(
    &mut self,
    seq_action: SequenceAction,
) -> Result<(), BundleFactoryError> {
    let seq_action_size = estimate_size_of_sequence_action(&seq_action);
```

```
        match self.curr_bundle.push(seq_action) {
            Err(SizedBundleError::SequenceActionTooLarge(_seq_action)) => {
                // reject the sequence action if it is larger than the max bundle
        size
                Err(BundleFactoryError::SequenceActionTooLarge {
                    size: seq_action_size,
                    max_size: self.curr_bundle.max_size,
                })
            }
            // [ ... ]
        }
    }
```

The issue here is that the user that submitted the transaction would end up seeing that it successfully submitted to the rollup node, but then it would never get executed.

Impact

If the `self.curr_bundle.max_size` is not set sufficiently high enough, this can occur for seemingly normal transactions and thus lead to a bad experience for the user.

Recommendations

Ideally, some type of an error should be returned to the user when they are submitting the transaction to the rollup node. This can be done by setting a transaction size limit on the rollup node so that such transactions cannot be submitted there.

Otherwise, this should be documented so that a user knows that they are able to manually submit such transactions to the Sequencer. It is only the composer that cannot submit such a transaction to the Sequencer.

Remediation

The client acknowledged this issue and stated that the `max_size` will be set to ~400KB, which is about the maximum size of a block on the Sequencer chain. They are aware of the impact on user experience from not returning a good error to the user, but they accept the risk because this limit will never be hit by legitimate transactions from legitimate users.

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. Suggestions for `astria-merkle`

While auditing the `astria-merkle` crate, we found some areas that could be improved in order to make the crate more ergonomic and less error-prone.

- The `Audit` struct itself should be marked with `#[must_use]` to ensure that the caller actually uses the struct. This will trigger a lint error for statements such as `proof.audit().with_leaf(leaf).with_root(root_hash);`, similar to `Audit::perform`.
- The `Tree::nodes` is currently a `Vec<u8>`, but everywhere that it is used, it is converted into a `Vec<[u8; 32]>` by slicing 32-byte ranges of it, and the length of the tree is considered 1/32 of the byte length. Instead, it could be stored as a `Vec<[u8; 32]>` to avoid the need for these conversions.
- The use of `Option` for the fields of `astria_merkle::{LeafBuilder,audit::LeafBuilder}` is unnecessary, and removing them eliminates some possible panics from the API.

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.

5.1. Component: Composer

The Composer's main responsibility is to watch a rollup node for transactions and compose them into transactions to be submitted to the Sequencer.

The Composer is mainly composed of a component called the Searcher, which itself has two subcomponents:

1. The Collector — Collects transactions from the rollup chain and sends them to the Executor.
2. The Executor — Bundles up transactions received from the collector and then executes them on the Sequencer chain.

Subcomponent: Collector

The Collector subcomponent watches rollup chains for transactions. Each transaction is then sent to the Executor subcomponent.

Currently, there is only one collector called the Geth Collector. This component subscribes to a Geth node and watches for new transactions. Each transaction is converted to a `SequenceAction` sent through to the Executor.

The excerpt of code that handles this is shown below:

```rust
while let Some(tx) = tx_stream.next().await {
    let tx_hash = tx.hash;
    debug!(transaction.hash = %tx_hash, "collected transaction from rollup");
    let data = tx.rlp().to_vec();
    let seq_action = SequenceAction {
        rollup_id,
        data,
        fee_asset_id: default_native_asset_id(),
    };

    match new_bundles
        .send_timeout(seq_action, Duration::from_millis(500))
        .await
        {
            // [ ... ]
        }
    }
```

Subcomponent: Executor

The Executor subcomponent receives `SequenceActions` from the Collector and attempts to bundle them up into a vector of Sequence actions. Once the bundle is considered full (i.e., it has reached the maximum size), it is submitted to the Sequencer.

Rollup transactions are rejected if they are over a certain `max_size` limit. In this case, "rejected" means that the transaction is never submitted to the Sequencer. This would mean that the user who made the transaction would see it submitted to the rollup node successfully, but then not be executed at all, which can be somewhat confusing. See Finding 3.14. ↗ for more information.

5.2. Component: Conductor

The Conductor's main responsibility is to connect the shared Sequencer to the execution layer, where the execution layer can be any node that executes transactions and returns an execution hash (for example, Geth).

The conductor receives blocks in two ways — either from the shared Sequencer directly or from the data-availability layer (i.e., Celestia). Blocks received from the shared sequencer are filtered for transactions that match the Conductor's rollup chain and then sent to the execution layer for execution.

Transactions received from the Sequencer are marked as "soft" as soon as they are executed. These transactions are not considered finalized until the same block is also fetched from Celestia, which means that Celestia acts as the ultimate source of truth in this scenario.

The three subcomponents of the Conductor are as follows:

1. The Executor
2. The Sequencer Reader
3. The Celestia Reader

Subcomponent: Executor

The Executor essentially does two things in parallel:

1. It receives soft blocks from the sequencer and executes the transactions in them immediately, and ensures that the block height matches the expected height. These executed blocks are then inserted into a list to await finalization.

2. It receives firm blocks from the data-availability layer (i.e., Celestia). These blocks are only executed if a corresponding soft block has not already been executed. Firm blocks are considered finalized.

For execution, the block is sent to the rollup execution layer node, which can be basically anything as long as it returns an execution hash. For example, a node like Geth or Erigon can be used for EVM-compatible transactions.

Subcomponent: Sequencer Reader

The Sequencer Reader component uses a Sequencer node's gRPC to fetch sequencer blocks. These blocks are sent through to the Executor marked as soft. If the executor channel is full, the blocks are scheduled and handled later.

Subcomponent: Celestia Reader

The Celestia Reader component uses an HTTP client to fetch blobs from Celestia periodically. Similar to the Sequencer Reader component, these blocks are sent through to the Executor, except that they are marked as firm. If the executor channel is full, the blocks are scheduled and handled later.

Additionally, this component also subscribes to receive block headers from Celestia using a WebSocket connection. These block headers are used to determine if there is a new block height available that should be fetched. This is essentially an optimization that helps this component fetch the latest blocks at all times.

Finally, if the subscription fails for any reason, it will trigger a resubscription to ensure the component is able to keep fetching block headers as needed.

5.3. Component: Merkle

Astria's Merkle tree implementation is used by the other components to prove inclusion of rollup IDs and transactions in blocks.

The `Tree` is a dense array of SHA-256 hashes, with even indexes being hashes of leaves and odd indexes being hashes of internal nodes. An implicit indexing scheme based on left-perfect binary trees is used, which permits appends with amortized $O(1)$ allocations and $O(\texttt{log}_2(n))$ rehashes. Leaves have a zero byte prefixed to the hash state, and internal nodes have one byte prefixed to their hash state.

The proof that some leaf data is included in the tree is a list of $\texttt{log}_2(n)$ hashes of its siblings going up to the root. An inclusion proof is verified by hashing the leaf data, hashing each pair of siblings (implicitly walking the tree), and comparing the resulting hash to the root.

5.4. Component: Sequencer Relayer

The Sequencer Relayer's main responsibility is to fetch blocks from the Sequencer and submit them to Celestia.

It contains a couple subcomponents:

1. The API server
2. The Relayer

Subcomponent: API Server

The API server has three routes:

1. `/healthz` — Returns whether the relayer state is healthy, which is true when the relayer is connected to both the Sequencer and Celestia.
2. `/readyz` — Returns whether the relayer state is "ready", which is true if the relayer currently has a block from the sequencer and a data-availability height from Celestia.
3. `/status` — Returns the `relayer_state` as a JSON object.

The `relayer_state` object looks as follows:

```
pub(crate) struct StateSnapshot {
    ready: bool,

    celestia_connected: bool,
    sequencer_connected: bool,

    latest_confirmed_celestia_height: Option<u64>,

    latest_fetched_sequencer_height: Option<u64>,
    latest_observed_sequencer_height: Option<u64>,
    latest_requested_sequencer_height: Option<u64>,
}
```

Subcomponent: Relayer

The Relayer's main responsibility is to fetch blocks from the Sequencer using gRPC and submit them to Celestia.

The submitter task is sent blocks asynchronously as they are fetched from the Sequencer. It converts `SequencerBlock` objects to `Blobs` that can be submitted to Celestia. This `Blob` structure is as follows:

```
pub struct Blob {
    /// A [`Namespace`] the [`Blob`] belongs to.
    pub namespace: Namespace,
    /// Data stored within the [`Blob`].
    #[serde(with
    = "celestia_tendermint_proto::serializers::bytes::base64string")]
    pub data: Vec<u8>,
    /// Version indicating the format in which [`Share`]s should be created
    from this [`Blob`].
    ///
    /// [`Share`]: crate::share::Share
    pub share_version: u8,
    /// A [`Commitment`] computed from the [`Blob`]s data.
    pub commitment: Commitment,
}
```

The converted `Blobs` are then submitted to Celestia using an HTTP client. It automatically retries submission for a while in case of unexpected failures, but ultimately it will quit after `u32::MAX` tries.

5.5. Component: Sequencer

The Astria Sequencer is a component that is shared across all rollup chains. It builds sequencer blocks out of transactions made on each rollup chain. It supports the following actions:

```
pub enum Action {
    Sequence(SequenceAction),
    Transfer(TransferAction),
    ValidatorUpdate(tendermint::validator::Update),
    SudoAddressChange(SudoAddressChangeAction),
    Mint(MintAction),
    Ibc(IbcRelay),
    Ics20Withdrawal(Ics20Withdrawal),
    IbcRelayerChange(IbcRelayerChangeAction),
    FeeAssetChange(FeeAssetChangeAction),
    InitBridgeAccount(InitBridgeAccountAction),
    BridgeLock(BridgeLockAction),
}
```

Each transaction on a rollup chain is defined by the following structure in the sequencer, where the `data` argument contains the transaction data, which can be in any format:

```
pub struct SequenceAction {
    pub rollup_id: RollupId,
    pub data: Vec<u8>,
    /// asset to use for fee payment.
    pub fee_asset_id: asset::Id,
}
```

In each sequencer block, there can be transactions from multiple rollup chains. It is the responsibility of the Conductor for each rollup chain to filter for transactions for their specific chain in order to execute and finalize them later on.

The Sequencer also supports a two gRPC queries:

1. `GetSequencerBlock` — Used to fetch a Sequencer block at a specific height.
2. `GetFilteredSequencerBlock` — Used to fetch Sequencer blocks that match a subset of rollup IDs. The returned block contains transactions and other metadata for those rollup chains.

Action: `Sequence`

The `Sequence` action is used by rollup chains to submit transaction data to the Sequencer. This data is then included into Sequencer blocks.

The definition of this action is as follows:

```
pub struct SequenceAction {
    pub rollup_id: RollupId,
    pub data: Vec<u8>,
    /// asset to use for fee payment.
    pub fee_asset_id: asset::Id,
}
```

In each sequencer block, there can be transactions from multiple rollup chains. It is the responsibility of the Conductor for each rollup chain to filter for transactions for their specific chain in order to execute and finalize them later on.

Action: `Transfer`

The `Transfer` action is used to transfer assets between accounts on the Sequencer chain. These assets are generally used to pay fees for other actions on the Sequencer chain. Its definition is as follows:

```
pub struct TransferAction {
    pub to: Address,
    pub amount: u128,
    // asset to be transferred.
    pub asset_id: asset::Id,
    /// asset to use for fee payment.
    pub fee_asset_id: asset::Id,
}
```

Action: `ValidatorUpdate`

The `ValidatorUpdate` action is used to notify Tendermint about any changes to the validator set. This is a sudo-only accessible action.

The definition of this action is as follows:

```
pub struct Update {
    /// Validator public key
    #[serde(deserialize_with = "deserialize_public_key")]
    pub pub_key: PublicKey,

    /// New voting power
    #[serde(default)]
    pub power: vote::Power,
}
```

Action: `SudoAddressChange`

The `SudoAddressChange` action is used to change the address of the `sudo` user. This action can only be executed by the current `sudo` user.

The definition of this action is as follows:

```
pub struct SudoAddressChangeAction {
    pub new_address: Address,
}
```

Action: `Mint`

The `Mint` action is used by the `sudo` user to mint Sequencer native assets to an arbitrary address.

The definition of this action is as follows:

```
pub struct MintAction {
    pub to: Address,
    pub amount: u128,
}
```

Action: `Ics20Withdrawal`

The `Ics20Withdrawal` action is used to withdraw an IBC asset from a source chain to a destination chain. This action performs checks to ensure the user has the required balance of the asset being transferred as well as the fee asset.

The definition of this action is as follows:

```
pub struct Ics20Withdrawal {
    // a transparent value consisting of an amount and a denom.
    amount: u128,
    denom: Denom,
    // the address on the destination chain to send the transfer to.
    destination_chain_address: String,
    // an Astria address to use to return funds from this withdrawal
    // in the case it fails.
    return_address: Address,
    // the height (on Astria) at which this transfer expires.
    timeout_height: IbcHeight,
    // the unix timestamp (in nanoseconds) at which this transfer expires.
    timeout_time: u64,
    // the source channel used for the withdrawal.
    source_channel: ChannelId,
    // the asset to use for fee payment.
    fee_asset_id: asset::Id,
}
```

Action: `IbcRelayerChange`

The `IbcRelayerChange` action is used by the `sudo` user to add and/or remove IBC relayers.

The definition of this action is as follows, where the `Addition()` enum is used to add a new relayer, while the `Removal()` enum is used remove a relayer:

```
pub enum IbcRelayerChangeAction {
    Addition(Address),
    Removal(Address),
}
```

Action: FeeAssetChange

The `FeeAssetChange` action is used by the `sudo` user to add and/or remove fee assets.

The definition of this action is as follows, where the `Addition()` enum is used to add a fee asset, while the `Removal()` enum is used remove a fee asset:

```
pub enum FeeAssetChangeAction {
    Addition(asset::Id),
    Removal(asset::Id),
}
```

Action: `InitBridgeAccount`

The `InitBridgeAccount` action is used to create a bridge account for a new rollup chain. This account can then be used as a bridge to the rollup chain using the `BridgeLock` action.

The definition of this action is as follows:

```
pub struct InitBridgeAccountAction {
    // the rollup ID to register for the sender of this action
    pub rollup_id: RollupId,
    // the assets accepted by the bridge account
    pub asset_ids: Vec<asset::Id>,
    // the fee asset which to pay this action's fees with
    pub fee_asset_id: asset::Id,
}
```

Action: `BridgeLock`

The `BridgeLock` action is used to transfer assets to the bridge account of a specific rollup chain. This effectively locks the assets into the bridge, allowing for the rollup chain to mint corresponding assets to the user's account on the rollup chain.

It is important to note that the `Transfer` action cannot be used to transfer tokens to a bridge account. Consequently, the `BridgeLock` action cannot be used to transfer tokens to a nonbridge account.

The definition of this action is as follows:

```
pub struct BridgeLockAction {
    pub to: Address,
    pub amount: u128,
    // asset to be transferred.
    pub asset_id: asset::Id,
    // asset to use for fee payment.
    pub fee_asset_id: asset::Id,
    // the address on the destination chain to send the transfer to.
    pub destination_chain_address: String,
}
```

Action: `Ibc`

The `Ibc` 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 `ConnectionOpen-Confirm` 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.

Message `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`

Message `ConnectionOpenTry` 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`) 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`

Message `ConnectionOpenAck` 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`), 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 comitted 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`, 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`

Message `ConnectionOpenConfirm` 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. 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. That is, 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), and
- 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`

Message `ChannelOpenTry` has B check

- that the connection to be established is exactly one hop, and
- 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 `ChannelEnd`s 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`

`ChannelOpenAck` has A check

- that its `ChannelEnd` is in `State::Init` or `State::TryOpen`,
- that its (A, B) `ConnectionEnd` is in `State::Open`, and
- 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`

`ChannelOpenConfirm` has B check

- that its `ChannelEnd` is in `State::TryOpen`,
- that its (A, B) `ConnectionEnd` is in `State::Open`, and
- 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`, 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`

Message `RecvPacket` 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, and
- 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, and
- 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, and
- 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 for production use.

During our assessment on the scoped Astria Shared Sequencer crates, we discovered 14 findings. One critical issue was found, 12 were of low impact, and the remaining finding was 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.