Security Audit Report

Interchain Security v.1.0: Provider Chain Safety

10.02.2023

Last revised 14.02.2023

Authors: Ivan Gavran, Andrey Kuprianov, Andrija Mitrovic

Contents

Audit overview

The Project

Interchain Security (ICS) is a feature that is about to be added to the Cosmos Hub. ICS allows smaller chains to take advantage of the Cosmos Hub strong security; these smaller chains are called *consumer chains* of the Hub, which is called the *provider chain*. ICS enables the Cosmos Hub validator set to be shared with consumers: the validators use their stake on the Hub to secure consumer chains. As a consequence, consumer chains do not have to establish their own validator sets, and the cost of attacking a consumer chain is the same as the cost of attacking the Cosmos Hub.

Technically, the provider chain and the consumer chains communicate via the IBC protocol: the provider sends updates on the validator set to consumers, and each consumer informs the provider of potential infractions committed on the consumer. In that way, the provider validator set is replicated[1] on every consumer chain.

It is important to note that a consumer chain can only be added through a governance proposal on the provider chain. If a proposal is accepted, validating on the consumer chain becomes mandatory for all members of the validator set.

Scope of this report

This is a report on the code review audit of the Interchain Security v1.0.0-rc1.

The audit took place from January 16, 2023 through February 10, 2023 by Informal Systems by the following personnel:

- Ivan Gavran
- Andrey Kuprianov
- Andrija Mitrovic

The audit was a part of a larger effort to make sure that ICS is ready for the Cosmos Hub upgrade. Before starting the work on this audit, Ivan Gavran and Andrey Kuprianov were engaged in providing security feedback on the ICS work-in-progress implementation.

In this audit, we focused our attention on the security of the provider chain with ICS enabled. We note that ICS is organized in a specific way: once added to the provider chain, it should not make any difference until the first consumer chain is added. Thus, in our analysis of the code, we always differentiate between three setups: the provider chain with ICS but no consumers, the provider chain with ICS and trustworthy consumers, and, finally, the provider chain with ICS and byzantine consumers (consumers that can behave in arbitrary ways, including buggy or malicious behaviors). We emphasize these scenarios because each consumer chain will be admitted in a separate governance proposal, likely receiving additional scrutiny. Regardless of the low likelihood of a completely arbitrary behavior, we flag potential issues that could arise in the case of byzantine consumers.

Conducted work

The audit team started its work on January 16th. The development team gave us an initial overview of the codebase, followed by more detailed walkthroughs of specific parts in the weeks that followed.

The audit team engaged in reviewing the code of different components of the ICS implementation, as well as examining the throttling protocol.

[1] A note on naming: the ICS feature is sometimes called *Replicated Security*, which is a more specific term than Interchain Security. However, we will use the term Interchain Security throughout this report to maintain consistency with the codebase name.

Timeline

- week 1: Code overview by the dev team, the audit team working on inspecting the system with no consumers.
- week 2: Walkthroughs of different features, the audit team inspecting the throttling mechanism and the addition of consumer chains.
- week 3: The audit team inspecting the exchange of VSC messages, slashing requests, and distribution of rewards.

Conclusions

Overall, we found the codebase to be of high quality: specification is complete and well-written, the code is well-structured and easy to follow, and the test suite includes unit tests, integration tests, and end-to-end tests. Some of the tests are based on a simplified implementation of the protocol, which functions as a model to compare the implementation to.

The main problem we found was an over-reliance on trustworthiness of consumers and not requiring any evidence of misbehavior. Based on that feedback, the development team implemented a reduction in consequences validators could face if a consumer chain reports their misbehavior, and is working on a release that would rely on evidence of misbehavior.

Despite the general high quality, we found some details that should be addressed in order to raise the quality of code and existing specification. One **High Severity** issue was found during this audit; the rest were marked Medium, Low or Informational severity.

Further Increasing Confidence

The scope of this audit was limited to manual code review and manual analysis and reconstruction of the protocols. We are aware that the ICS team ran a separate protocol audit in which the protocol's TLA+ model was checked by a formal verifier. To further increase confidence in the implementation, we recommend augmenting the existing test suite with TLA+ model-based adversarial testing. This would also make it easier to maintain the test suite in the event of future changes to the codebase.

It is our understanding that the Interchain Security team intends to pursue such measures to further improve confidence in their system.

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Golang
- **Artifacts**:
  - ICS28 Cross Chain Validation: 94baa3f
  - Interchain Security: v1.0.0-rc1
  - Cosmos SDK: v0.45.11-ics, specifically the changes wrt v0.45.11

Engagement Summary

- **Dates**: 16.01.2023 to 10.02.2023
- **Method**: Manual code review & protocol analysis
- **Employees Engaged**: 3

Severity Summary

Finding Severity	#
Critical	0
High	1
Medium	2
Low	3
Informational	2
Total	**8**

System Overview

Interchain Security introduces an option for new or existing blockchains to lease security from a larger blockchain. The (smaller) blockchains that lease security are called *consumer chains*, and the (larger) blockchain that provides security is called the *provider chain*. Validators of the provider chain will additionally validate on consumer chains using their stake from the provider. In effect, misbehavior on any of the consumers becomes equally costly as misbehavior on the provider itself. Importantly, a consumer can only be added through a governance proposal on the provider.

In this section, we give a high-level overview of the system, which will be useful for understanding the rest of the report. For more details on the system, see the specification file or the documentation.

In order to achieve the replication of security, the provider (P) and a consumer (C) are exchanging the following messages over IBC:

- $P \rightarrow C$: **Validator Set Change (VSC)**. This message makes sure that the consumer chain keeps up with the changes of the validator set on the provider chain.
- $C \rightarrow P$: **VSC matured**. This message is sent once the consumer's unbonding period has elapsed upon receiving the VSC. Without receiving this message, the provider chain will not unbond the relevant stake (because that stake is still used on the consumer).
- $C \rightarrow P$: **Slash requests**. Upon noticing misbehavior of a validator, the consumer will send a request to the provider to slash the stake of the misbehaving validator, and jail or tombstone the validator.
- $C \rightarrow P$: **Rewards**. Periodically, the consumer transfers reward tokens to the distribution account of the provider chain. These tokens have been accumulated in a separate account on the consumer chain at the beginning of every block.

The first three messages are sent over the same, ordered channel. The last message, which sends rewards to the provider, is sent over a separate, unordered channel.

Note that there is no assumption of consumers and the provider being synchronized. In the next subsection, we will go into more detail on how the exchanged messages provide a necessary synchronization mechanism, as well as the block structure of both the provider and consumers.

Block Structure

Let us dive deeper into how the provider and a single consumer are interacting. We will use Figure 1 as a schematic representation of the interaction. On the left side of the figure (in blue) we see the provider chain, and on the right side (in red) we see the consumer chain. Two blocks of both chains are shown. (As mentioned earlier, blocks can be produced at arbitrary speeds, and the two shown blocks are not necessarily consecutive.) Each block is separated into three parts: the upper part denotes the `BeginBlock` method, the middle part denotes handling of transactions (`DeliverTx` method), and the lower part denotes the `EndBlock` method.

The messages of the ordered channel are denoted by arrows. (The messages of the unordered transaction channel are not shown in the figure.)

In the provider's `BeginBlock` method, updates to the validator set are applied. These updates will be sent to the consumer in the `EndBlock`. You will notice that they are first enqueued and only then sent to the consumer. This is a common pattern in the communication scheme and it makes sure that if the consumer's channel is not yet established, the updates get preserved and are sent only once the consumer is ready to receive them. (The figure uses a shorthand `enQ` for `enqueue`.)

Prior to enqueueing and sending the updates to the consumer, the `EndBlock` also slashes the stake of misbehaving validators. It does so per the consumer's report. Importantly, it does not require proof of misbehavior from the consumer.

Trusting the consumer chain to report misbehavior faithfully is listed as one of the assumption in the specification (see Evidence Provision). Regardless of the Evidence Provision assumption in the protocol specification, the

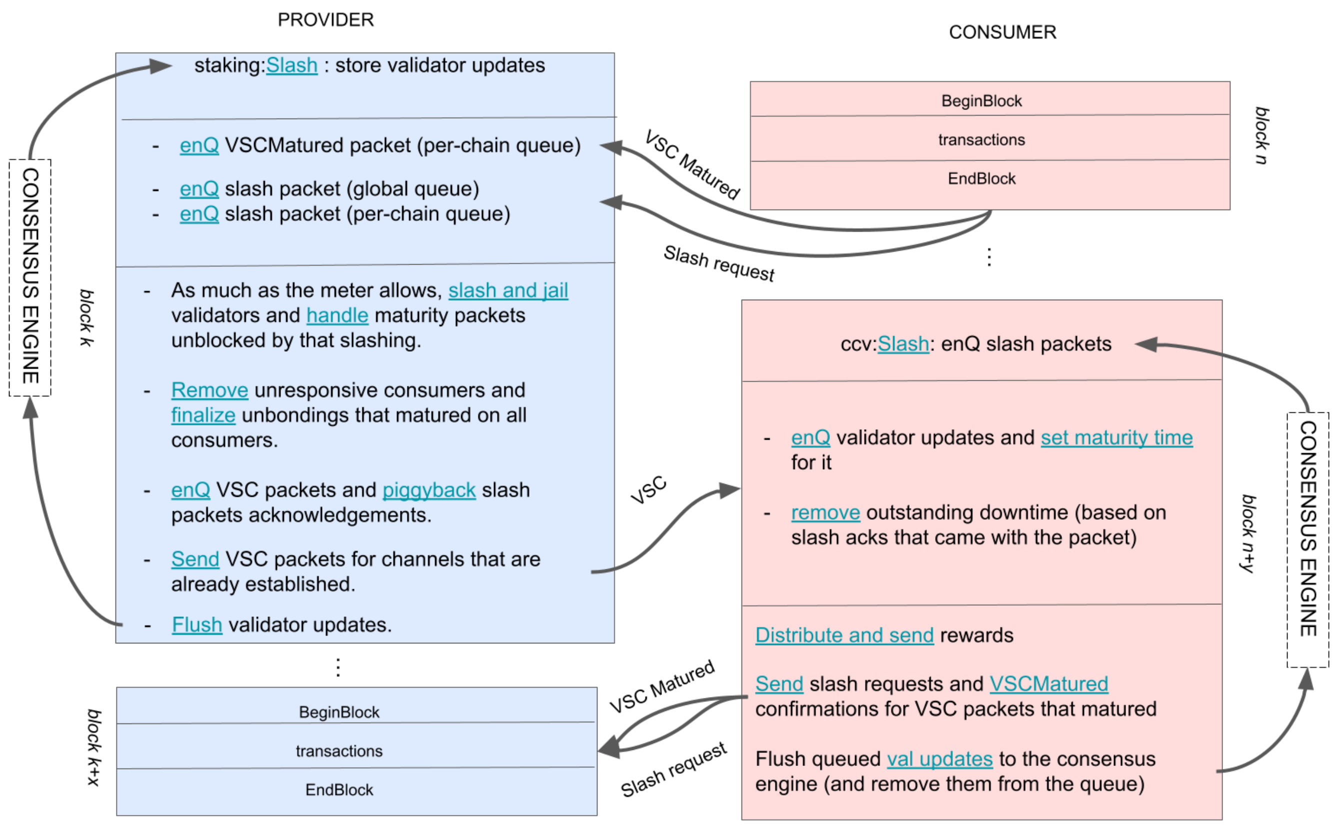
PROVIDER
CONSUMER
staking:Slash : store validator updates
- enQ VSCMatured packet (per-chain queue)
- enQ slash packet (global queue)
- enQ slash packet (per-chain queue)
- As much as the meter allows, slash and jail validators and handle maturity packets unblocked by that slashing.
- Remove unresponsive consumers and finalize unbondings that matured on all consumers.
- enQ VSC packets and piggyback slash packets acknowledgements.
- Send VSC packets for channels that are already established.
- Flush validator updates.
CONSENSUS ENGINE
block k
BeginBlock
transactions
EndBlock
block k+x
VSC Matured
Slash request
block n
VSC
ccv:Slash: enQ slash packets
- enQ validator updates and set maturity time for it
- remove outstanding downtime (based on slash acks that came with the packet)
Distribute and send rewards
Send slash requests and VSCMatured confirmations for VSC packets that matured
Flush queued val updates to the consensus engine (and remove them from the queue)
block n+y
CONSENSUS ENGINE

Figure 1: Schematic representation of the interaction between the provider and a consumer

implementation includes a mechanism to prevent a byzantine consumer from removing a large number of validators at once. This is why the figure says that validators should be removed "*As much as the meter allows*". The whole mechanism is described in the section Throttling overview of this report.

The provider also receives transactions notifying it that some VSCs have matured on the consumer. When they are received, these notifications are enqueued and processed in the `EndBlock`. If maturity notifications are received from all existing consumers, the provider informs the staking module that relevant unbonding operations can be finalized. If, however, a maturity notification for a VSC is not received for too long (the default timeout is 5 weeks), the provider will remove the consumer from the set of active consumers. This ensures that unbonding is not blocked indefinitely.

Finally, when sending VSCs to the consumer in the `EndBlock`, the provider also sends acknowledgments for all received slash requests.

On the consumer's side, once the updates on validators' powers (VSC packets) are received from the provider, they are enqueued (to be processed later in the `EndBlock`) and the maturity time is set for the VSC packet (depending on the consumer's unbonding time). Moreover, the acknowledgments for the slash requests are used to unset the boolean flag `outstandingDowntime`, which is used to prevent the consumer from sending multiple slash requests for the same downtime infraction.

The enqueued updates on validators' powers are returned to the consensus engine in the `EndBlock` method. On top of that, `EndBlock` is responsible for sending slash requests (as enqueued in the `BeginBlock`) and for sending notifications that some VSCs have matured. Finally, the `EndBlock` method also accumulates rewards and sends them to the provider every `period` blocks (where the default value for `period` is `1000`). The rewards are sent directly to the provider's distribution module so there is no separate handling of rewards on the provider's side introduced by the Interchain Security module.

Thus far, we have given an overview of the standard communication scheme between the provider and a consumer. The following subsections will cover additional features of Interchain Security:

- **adding and removing consumers**: consumers are introduced to Interchain Security (and removed from it) using governance proposals on the provider.
- **reward distribution**: rewards from consumers are distributed to validators.
- **throttling**: a large number of validators cannot be removed at once.

Adding and removing consumer chains

A consumer chain can be added to the provider chain only through a governance proposal. Removing a consumer chain can be triggered by a governance proposal or by various validity checks.

Proposal processing

The addition and removal of consumer chains through proposals are done in two steps:

1. Gathering proposals, verifying them, and adding them to a queue of pending proposals.
2. Going through the queue of pending proposals at the beginning of each block and executing them.

There are two types of proposals for the addition and removal of consumer chains:

1. `ConsumerAdditionProposal` is a governance proposal on the provider chain to spawn a new consumer chain. First, a consumer addition proposal is executed in a cached context for verification. Here, a consumer client is created if it does not already exist for the proposed consumer. After the verification, cached writes are discarded. If the verification is successful, a pending consumer addition proposal is stored. If there are multiple addition proposals for the same chain, only the last will be stored.
2. `ConsumerRemovalProposal` is a governance proposal on the provider chain to remove/stop a consumer chain. First, the consumer removal proposal is executed in a cached context for verification. After the verification, cached writes are discarded. If the verification is successful, a pending proposal for removal is set for that consumer chain.

Addition of consumer chains

The addition of consumer chains through a proposal is done at the beginning of each block in the `BeginBlockInit` function. First, addition proposals whose spawn time is not before the current block time are acquired. Then, for each proposal, the creation of a consumer client in a cached context is done. If this passes without errors, then the cached context is written to the context. In the end, executed addition proposals are deleted from the pending list (the deletion happens whether the proposal was executed successfully or not).

Removal of consumer chains

As mentioned before, removing consumer chains can be done in a few ways:

- Removing consumer chains through proposal is done at the beginning of each block in the `BeginBlockCCR` function. First, removal proposals whose stop time is not before the current block time are acquired. Then, for each proposal, stopping the consumer chain in a cached context is done. If this passes without errors, then the cached context is written to the context. Note that for every stopped chain, the state is cleaned for the given consumer chainID and the outstanding unbonding operations are completed. In the end, executed removal proposals are deleted from the pending queue.
- The consumer chain can be removed/stopped if a packet from the provider to the consumer times out with a reason that isn't an unknown channel.
- The consumer is removed/stopped if the channel still exists and the acknowledgment VSC packet from the consumer could not be successfully decoded.
- The consumer chain can be removed at the end of each block for two reasons:
  - The consumer is not initialized on time.
  - Consumers for whom the maturity notification for a VSC packet was not received before the timeout.

Reward distribution overview

The process of validating transactions on the consumer side is rewarded. Distribution of rewards is a process of splitting and distributing the consumer block rewards between the consumer chain itself and the provider chain. Distribution of rewards occurs at the end of each block on the consumer side. Rewards for the provider are accumulated to be transmitted to the provider. Transmissions occur after a predefined number of blocks passes after the last transmission. This predefined number of blocks is kept track of in the store under `types.KeyBlocksPerDistributionTransmission`. This process is schematically presented in Figure 2.

Each consumer has three accounts for the rewards which are used for distribution:

- **Consumer-chain fee collector account**: an account used for collecting fees on the consumer. These will be split into two parts and sent to the following two accounts at the end of each block.
- **Consumer-redistribution account**: an account used for collecting rewards that should be redistributed on the consumer chain.
- **Consumer to send to provider account**: an account used as a buffer for collecting rewards that should be sent to the provider chain.

Distribution process consists of two steps on the consumer side:

1. **Distribute rewards internally**: In this step, the rewards for the current block are first split into two parts. The first part is the amount that should be kept on the consumer chain. The second part is accumulated in a "buffer", where it waits to be sent to the provider.
2. **Send accumulated reward to the provider chain**: When the number of blocks for transmission has passed, the accumulated rewards are sent to the provider. If the sending times out or fails, the sent amount will be returned to the buffer and wait for the next transmission.

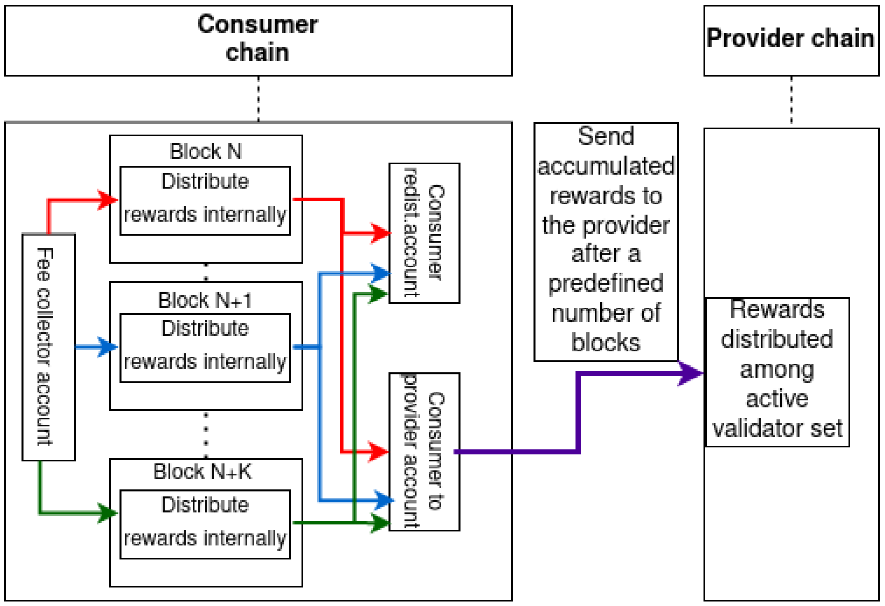
Consumer chain
Provider chain
Block N
Distribute rewards internally
Block N+1
Distribute rewards internally
Block N+K
Distribute rewards internally
Fee collector account
Consumer redist.account
Consumer to provider account
Send accumulated rewards to the provider after a predefined number of blocks
Rewards distributed among active validator set

Figure 2: Schematic representation distribution of rewards

At the provider side, the rewards from the consumer will arrive periodically. When the rewards arrive, they will be added to the amount that should be split among the validators. Note that this reward is split among the active validator set, which can differ from the set of validators that earned that reward. This can happen because validator set can change before the rewards are transmitted.

Throttling overview

ICS currently does not check claims of infractions submitted by consumer chains. Without additional protection, this opens up a possibility of a byzantine consumer chain jailing or tombstoning a large fraction of power of all validators, allowing collaborating validators to take full control of the chain. To mitigate this threat, ICS throttles the rate at which consumer chains can slash validators: only validators whose total power is below a predefined fraction of the total validator power may be slashed per predefined period. This extra time allows validators and the community to appropriately respond to the attack if they notice that validators are being slashed inappropriately.

More detailed overview with described technical terms and parameters can be found here.

Throttling mechanism algorithm

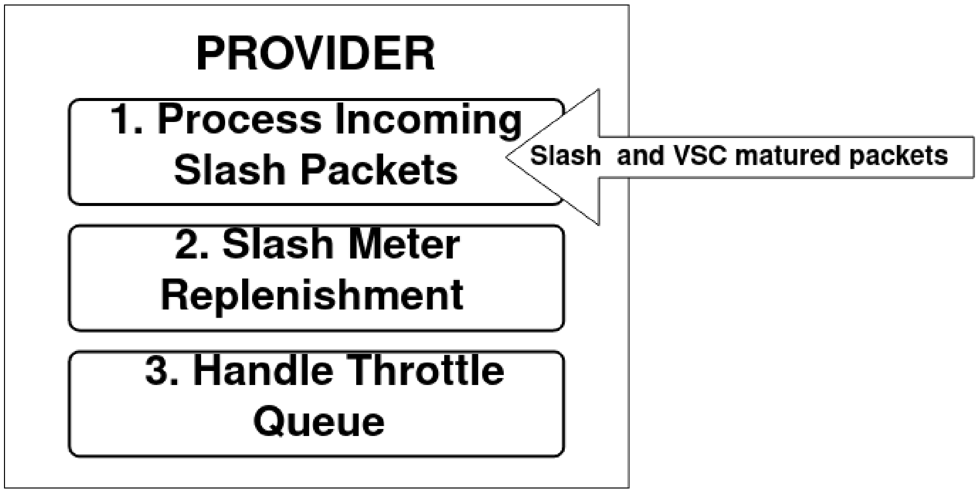
PROVIDER
1. Process Incoming Slash Packets
2. Slash Meter Replenishment
3. Handle Throttle Queue
Slash and VSC matured packets

Figure 3: Throttle mechanism steps

The throttling mechanism consists of three steps:

1. **Process Incoming Slash Packets**: Slash request packets are gathered along with VSC maturity packets (in order) into queues per consumer chain. Separately, only slash packets are enqueued into the global entry queue.
2. **Slash Meter Replenishment**: The fraction of the total validator power that can be slashed is defined by the current value of the slash meter. Before processing the upcoming slash packets, a check if the meter needs to be replenished is performed, provided that the replenishment period has elapsed. Replenishment of the meter is done as follows, which is schematically presented in Figure 4:
   - Replenish the slash meter with the value of allowance if the current time is equal to or after the current replenish time. At the same time, set the next replenish time by adding replenish period time to the current block time.
   - Check that the slash meter is not greater than or equal to the allowance for the current block. This can occur if the total voting power of the provider chain has decreased since previous blocks. In that case, it is set to the maximal possible fraction of the total voting power.
3. **Handle Throttle Queue**: As long as the meter is not negative, slash packets are taken from the global queue (with meter decreased by the power of the validator that is being slashed) alongside the set of trailing VSC maturity notifications are processed. This step is schematically presented in Figure 5.

Time analysis for processing maturity request

The global entry queue consists of sequences of slashing requests $(S_1, S_2, ...)$. Per-chain queues additionally include maturity notifications. Since maturity notifications do not influence processing time at all (any maturity notifications-prefix of the queue is processed immediately), we can think of a queue $(S_1, S_2, \dots, S_n, M)$, where M is the maturity notification whose delay interests us. (In the implementation, M is a part of a per-chain queue and won't be processed until the S_n, a slash request of the same chain, is processed.)

Let $t(M)$ be the time needed to process maturity notification M. (If there were no throttling, that would happen as a part of the same block, and we would consider that to be $t(M) = 0$.)

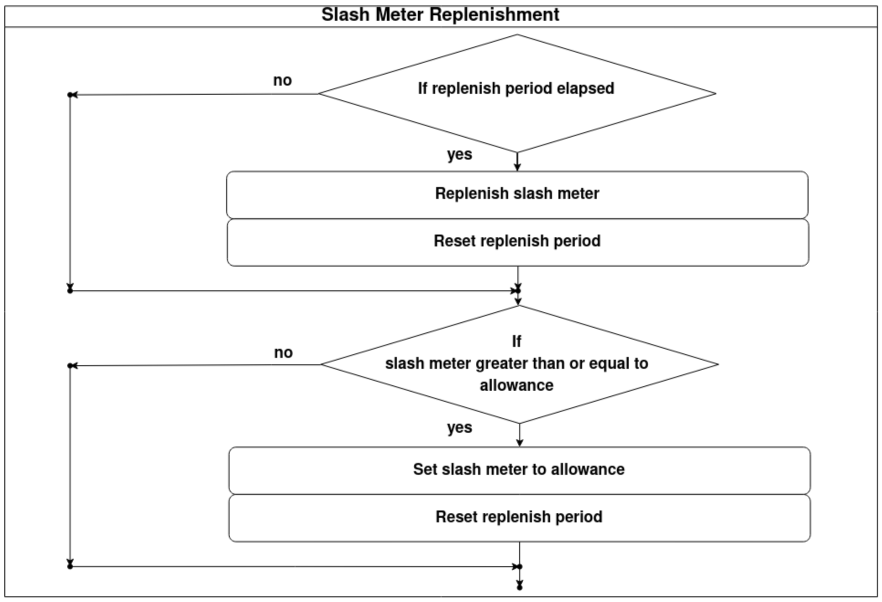
Slash Meter Replenishment
If replenish period elapsed
no
yes
Replenish slash meter
Reset replenish period
If slash meter greater than or equal to allowance
no
yes
Set slash meter to allowance
Reset replenish period

Figure 4: Slash Meter Replenishment

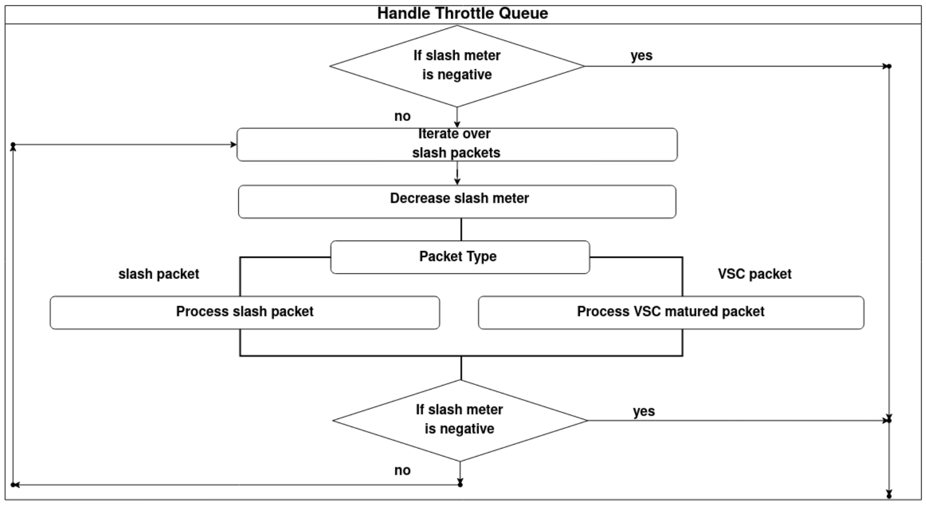
Handle Throttle Queue
If slash meter is negative
yes
no
Iterate over slash packets
Decrease slash meter
Packet Type
slash packet
VSC packet
Process slash packet
Process VSC matured packet
If slash meter is negative
yes
no

Figure 5: Handle Throttle Queue

At block b, with the full meter, we can process requests $S_1, ..., S_j$ which satisfy the inequality $S_1 + \ldots + S_{j-1} < f \cdot totalPower(b)$, where:

- f is the `SlashMeterReplenishFraction` parameter
- $totalPower(b)$ is the total power of validators at block b, as obtained by the function `GetLastTotalPower`

We note here that there is some freedom with respect to the value of the S_j request. However, in the long run, this is averaged out since the meter can go negative and will take time proportional to the value of S_j to get back to the positive state.

Let T_{min} be the minimum total power of the chain over all blocks in which the sequence $S_1, \ldots, S_n$ is processed.

Then, whenever the meter is full, we can process at least $S_1, ..., S_j$ which satisfy the inequality $S_1 + \ldots + S_{j-1} < f \cdot T_{min}$. Thus, the overall time needed to process the maturity notification M is

$$t(M) \leq \frac{\sum_{i=1}^{n} S_i}{f \cdot T_{min}} \cdot p$$

where p is `SlashMeterReplenishPeriod`. This number is given in the units of `SlashMeterReplenishPeriod`.

With current default values of $f = 0.05$ and $p = 1$ hour, we will process M after no more than $20 \cdot \frac{\sum_{i=1}^{n} S_i}{T_{min}}$ hours.

Overview of Cosmos SDK changes

Here we review the Cosmos SDK changes due to ICS introduction via comparing the branches v0.45.11 vs v0.45.11-ics.

Identification and reference counting for unbonding operations

Interchain Security needs to make validators and delegators responsible for infractions committed on consumer chains. For that purpose, unbonding on provider needs to take into account not only the provider unbonding period, but also the unbonding that happens on consumer chains; this entails a few necessary changes in the Cosmos SDK.

The most important data structures related to unbonding now contain two additional components, `uint64 unbonding_id` and `int64 unbonding_on_hold_ref_count`:

- UnbondingDelegationEntry
- RedelegationEntry
- Validator: contains not a single, but a list of multiple `unbonding_ids`

The `UnbondingID` is a unique 64-bit integer, that is kept track of in the store under `types.UnbondingIdKey`, and is supposed to uniquely identify every unbonding operation: undelegation, redelegation, or validator unbonding. The `UnbondingID` is retrieved and auto-incremented in the function IncrementUnbondingID. The function is called from the three places that create or update the above datastructures upon an unbonding operation:

- SetUnbondingDelegationEntry()
- SetRedelegationEntry()
- BeginUnbondingValidator()

The `UnbondingOnHoldRefCount` is a reference counter that can be incremented in order to put any unbonding operation "on hold"; there is an API for that provided by the Staking module to external modules:

- the hook AfterUnbondingInitiated(ctx, id) will be called when any unbonding operation is initiated
- the function PutUnbondingOnHold(ctx, id) allows an external module to put the operation with the given `id` on hold by incrementing the reference counter; this is used in order to wait until the unbonding operation completes on all connected consumer chains.
- the function UnbondingCanComplete(ctx, id) allows an external module to "release" the previously obtained hold via decrementing the reference counter, and permitting the unbonding to proceed in case the counter reaches zero.

All unbonding operations that would previously automatically complete after the unbonding period finishes (currently 3 weeks for Cosmos Hub), now also require that the operation is released (not on hold anymore) by all previously issued hold requests (see `PutUnbondingOnHold` above). Technically, this is done in the following places:

- the function CompleteUnbonding() checks for each unbonding delegation entry that it's not `OnHold`
- the function CompleteRedelegation() checks for each redelegation entry that it's not `OnHold`
- the function UnbondAllMatureValidators() checks that the reference counter for a validator in the unbonding queue has reached zero.

`InfractionType` is introduced in the signatures of `Slash` functions

The InfractionType enum is added to the SDK, and it is introduced as the last argument in the signatures of Staking keeper Slash() function and Slashing keeper Slash() function.

This change touches on a lot of files, but doesn't seem to introduce any additional functionality of complexity.

ABCI Validator updates are tracked for each block

Interchain security, in order to function correctly, needs to retrieve the changes to validator sets, which it then propagates to the consumer chains. For that, the following changes are introduced into the Cosmos SDK:

- the new Protobuf message ValidatorUpdates is introduced
- Staking keeper API is extended with two functions: SetValidatorUpdates(), and GetValidatorUpdates(), which, respectively, set and retrieve the latest ABCI validator updates in the store
- the function ApplyAndReturnValidatorSetUpdates() is extended to store ABCI validator updates in the store.

It should be noted that, although validator updates are stored for each block, this should not pose significant overhead, as only the changes to the current validator power distribution are tracked.

Threat Inspection

In the present section, we outline our analysis of the main threats that might materialize with the introduction of Interchain Security (ICS), and how, in our opinion, these threats are mitigated (or not) in the audited codebase; in the present audit report **we focused only on threats wrt. the provider chain**. It should be noted that, while we do our best in the analysis below, **a security audit can by no means guarantee the absence of threats**: the real guarantees can be provided only via systematic quality assurance, including manual and automated testing, and, ultimately, via the usage of formal methods, such as model-based testing or formal verification.

We have identified three main categories of threats whose probability might increase when ICS comes into operation:

- **Halting of the provider chain**. (*Impacted users: all provider token holders*) As many of ICS operations are performed in BeginBlockers or EndBlockers, and some contain panics, triggering of any of these panics would halt the provider chain. We took the effort to inspect whether and how these panics under different conditions.
- **Delaying or not completing unbonding operations on the provider chain**. (*Impacted users: provider token holders, performing unbonding operations*) ICS introduces additional logic into Cosmos SDK to allow delaying unbonding operations on the provider chain, until they complete on all connected consumer chains. We have inspected this modified logic, as well as how it interacts with the core ICS functionality on both the provider and consumers.
- **Harming validators**. (*Impacted users: provider validators*) ICS brings with itself more requirements for validators: now they need to validate both on the provider and on the consumer chains, and these activities interact in non-trivial ways. We have inspected whether and how honest validators could be slashed/jailed/tombstoned in this new setup, and what measures are already in place, or should be introduced in order to prevent this from happening.

Finally, as ICS is a complex project, and it is planned to be gradually put into operation over the course of the year 2023, we undertook a more granular approach, and inspected the above threats under these conditions:

- **No consumers**. This is the first phase, ICS release 1.0, as it is planned to be introduced into Cosmos Hub. Only the core ICS provider functionality, as well as the changes to Cosmos SDK will be present, but no consumer chains will be allowed yet.
- **Trustworthy consumers**. This is the scenario that is planned to be introduced with ICS 1.1, and a few first consumer chains are going to be connected. These consumer chains are going to receive the highest level of public attention, as well as the high quality security audits are going to be performed for them, so these first chains can be expected to strictly follow the ICS protocols.
- **Byzantine consumers**. As more and more consumer chains are going to be connected to ICS, the level of achievable security guarantees wrt. consumers is likely to decrease, and thus the provider chain should be ready to handle byzantine consumers, i.e. those that might exhibit arbitrary (byzantine) behavior.

The results of the threat inspection are presented in Table 2. It should be noted that the analysis is done wrt. the codebase under audit, and doesn't account for many improvements that are already being implemented.

Table 2: Threat landscape under different conditions

Threat \ conditions	No consumers	Trustworthy consumers	Byzantine consumers
Provider halts	Low risk	Medium risk	Medium risk
Unbonding delayed	Low risk	Low risk	Medium risk
Validators harmed	Low risk	Low risk	High risk

We note that the codebase has received a number of changes since this audit. The introduced changes are changing the threat landscape that is presented in the Table 2. We have not re-audited the codebase with the new changes, but here we give an overview of release candidates that followed the audit, and, where relevant, comment on their **expected** (but not audited) impact on the threat landscape.

- v1.0.0-rc2: Code re-organization. No expected impact on the threat landscape.
- v1.0.0-rc3: IBC is bumped to version `4.2.0` (from previous `3.4.0`). No expected impact on the threat landscape.
- v1.0.0.-rc4: Testing and documentation improved, the problem described in IF-ICS-02 is fixed. **With this change, the throttling mechanism works as expected and makes it more difficult for malicious actors to take over the provider validator set.**
- v1.0.0-rc5: Consumer initiated slashing and tombstoning is disabled (only jailing remains). **This change significantly reduces the threat to validators under existence of byzantine consumers.**
- v1.0.0-rc6: Testing and documentation are improved, SDK version is bumped to `v0.45.13-ics` (from the audited `v0.45.11-ics`). Governance proposal is added that allows slashing and tombstoning a validator for double signing or other serious infractions on the consumer chains. (This replaces previously existing automatic consumer-initiated slashing.) No expected impact on the threat landscape.
- v1.0.0.-rc7: Documentation is improved. No expected impact on the threat landscape.

Threat: Provider halts or its state is corrupted

In this section, we inspect the possibility of the provider halting due to panics in `BeginBlock` and `EndBlock`. We will examine three different settings: a) with no consumers added, b) with trustworthy consumers (those following the specification), and c) with byzantine consumers.

No consumers

The case with no consumers (active or pending) is the least interesting, but also the first concrete state of the system once Interchain Security is deployed. We inspect the properties of `BeginBlock` and `EndBlock` functions and examine if they introduce, when deployed prior to any consumers joining, any side effects.

Property 1 `BeginBlock` neither panics nor changes the state of the provider chain.

We examine the same property for functions `BeginBlockInit` and `BeginBlockCCR`.

The property holds for `BeginBlockInit` since (in the absence of pending proposals) the slice `propsToExecute` is going to be empty. Then, the iteration over it is a no-op, and the same is true for the iteration in the function `DeletePendingConsumerAdditionProps`.

To see why `propsToExecute` is an empty slice, note that `GetConsumerAdditionPropsToExecute` iterates over stored values with the prefix `[]byte{types.PendingCAPBytePrefix}`. The writes under keys with such a prefix happen only after a proposal was handled, or when setting up the chain reading from the genesis file. However, our assumption was that there was no proposal.

A fully analogous analysis holds for why `BeginBlockCCR` does not panic or change the state. In this case, the inspected key prefix is `PendingCRPBytePrefix`.

Property 2 `EndBlock` does not panic and it minimally changes the state of the provider chain.

We examining related properties for three functions: `EndBlockCIS`, `EndBlockCCR`, and `EndBlockVSU`.

Property 2-a `EndBlockCIS` does not panic and changes the state only for the throttling meter-related variables.

Inside `CheckForSlashMeterReplenishment` → `GetLastSlashMeterFullTime`, there is a potential panic if the key `providertypes.LastSlashMeterFullTimeKey()` is not found in the store. However, it won't happen since that key is set in `InitGenesis` → `InitializeSlashMeter` → `SetSlashMeter`. This function might set the slash meter variable (stored under `providertypes.SlashMeterKey()`) by calling in `SetSlashMeter` in case the total validators power changes and thus makes the meter value greater than its allowance. (At the same time the value under the key `providertypes.LastSlashMeterFullTimeKey()` is set.) The sanity-check panics inside `SetSlashMeter` will not be triggered because the value of the meter is set to the allowed value by calling `GetSlashMeterAllowance`.

`HandleLeadingVSCMaturedPackets` is an empty iteration because it iterates over `GetAllConsumerChains`, which is empty in the absence of consumer chains (with an argument similar to the one in Property 1).

Inside the `HandleThrottleQueues` method, there is an iteration over the result of `GetAllGlobalSlashEntries`. This too is empty because it reads from the key `GlobalSlashEntryBytePrefix`, which is only set from the `OnRecvSlashPacket` (→ `QueueGlobalSlashEntry` → `GlobalSlashEntryKey` → `GlobalSlashEntryBytePrefix`). With no consumer chains, it will never be set. This is followed by deletion over the empty slice (which is again an empty iteration). Finally, there is a call to `SetSlashMeter`. This function has two potential sanity-checks panics. But because the function was called with the unchanged `meter` as its second argument (the value of which was set to allowed values inside `CheckForSlashMeterReplenishment`), these sanity checks won't panic.

Property 2-b `EndBlockCCR` neither panics nor changes the state. The function consists of iterations over return values of functions `GetAllInitTimeoutTimestamps` and `GetAllChannelToChains`.

The first one reads from the key `InitTimeoutTimestampBytePrefix`, which is only set in `CreateConsumerClient` (and this is the only path to set the desired key), a function called only if there are existing consumer proposals.

The second one, `GetAllChannelToChains`, reads from kyes prefixed with `types.ChannelToChainBytePrefix`, which are only set in the genesis file or upon `OnChanOpenConfirm`.

Property 2-c `EndBlockVSU` does not panic and it only changes the state under the key `types.ValidatorSetUpdateIdKey()`.

The function consists of three parts. This first one, `completeMaturedOnbondings` is a no-op because it iterates over `ConsumeMaturedUnbondingOps`, which is empty. (Indeed, besides the genesis file, the matured unbonding ops are updated in `HandleVSCMaturedPacket`. With no consumers, this function is never called.) The second part, a function `QueueVSCPackets` is a no-op because it iterates of `GetAllConsumerChains`. After the iterations, the validator set update id (stored under `types.ValidatorSetUpdateIdKey()`) is incremented. Finally, the third part, a function `SendVSCPackets` is a no-op because it also iterates over (empty) `GetAllConsumerChains`.

Trustworthy consumers

With only trustworthy consumers present in the system, we want to inspect if it is ever possible for the provider to be halted. To that end, we examined all potential panics originating from `BeginBlock` or `EndBlock`.

We found that most of those panics are raised in cases of problems with marshalling or unmarshalling data, or when performing sanity checks (e.g., a value not found in the store which should be there, internal state corrupted).

There are two exceptions worth mentioning. The first one is the panic raised upon a failed attempt to send a VSC packet. There, if an error is of the kind `ErrClientNotActive`, it is handled, and the panic is raised otherwise. We provided more details on this issue in the Finding IF-ICS-03.

The second one is the panic inside `SetThrottledPacketDataSize`, raised when the throttling queue goes above its maximum allowed size. This is not a problem because `SetThrottledPacketDataSize` is called from the `EndBlock` method only when items are taken out of the throttling queue. Adding them to the queue is attempted when handling a packet. Thus, the queue size will never go over its maximum allowed value.

Finally, we would like to flag the possibility of the provider spending all its gas budget during `BeginBlock` and `EndBlock`. This scenario is described in Issue 726, found independently of this audit.

Byzantine consumers

With the analysis of panics from the previous section, we conclude that the additional freedom in a consumer's behavior cannot cause the provider to halt.

Threat: Unbondings are delayed or don't complete

In this section, we examine the threat to the holders of tokens on the provider chains, wishing to perform an unbonding operation, that the operation is either delayed (compared to non-ICS case), or doesn't complete.

No consumers

With Interchain Security, the `Staking` module will call into the hook `AfterUnbondingInitiated`, with the purpose of putting the unbonding on hold until the unbonding period has passed on all consumer chains. When there are no consumer chains, however, the hook returns prematurely and does nothing. Thus, effectively, unbonding operations are never put on hold in case of absent consumers, and the provider unbonding operations function as before.

Trustworthy consumers

With consumers present, every unbonding operation on the provider will be put on hold, and wait for all unbonding operations to complete on the consumer chains, connected at the moment when this unbonding operation was taking place. When consumer chains are following the ICS protocol, unbonding period on the consumers should be less than that on the provider, e.g. in interchain-security/docs/params.md it is said that:

> The `ConsumerUnbondingPeriod` is set via the `ConsumerAdditionProposal` gov proposal to add a new consumer chain. Unbonding operations (such as undelegations) MUST wait for the unbonding period on every consumer to elapse. Therefore, for an improved user experience, the `ConsumerAdditionProposal` on every consumer chain SHOULD be smaller than `ProviderUnbondingPeriod`.

Thus, a *live* consumer chain following the protocol should finish unbonding *before* the unbonding would finish on the provider, and unbonding operations on the provider won't be delayed. There is a small problem we've detected, in that the implementation doesn't enforce the above requirement (see the finding IF-ICS-04); but for the problem to arise a misconfigured consumer addition governance proposal should pass, and the probability of this happening is very low.

We have analyzed if the throttling queue could cause unbonding delays. As discussed in Section Throttling overview, the throttling queue is used to prevent the provider from sending too many packets to the consumer chains, and thus causing them to halt. Processing of a VSC maturity packet M that arrived after slash requests $S_1, S_2, \ldots, S_n$ will take no more than $20 \cdot \frac{\sum_{i=1}^{n} S_i}{T_{min}}$ hours. The fraction in this inequality is, for all practical scenarios, significantly smaller than 1, and thus the possible delay is negligible.

Another possibility for the unbondings to be delayed, is when either a consumer chain stops to function correctly, or the IBC communication ceases; moreover, any of these events need to maintain their (non-live) state for a considerable amount of time (~ 3 weeks), which we consider to be outside of boundaries of trustworthy behavior.

Byzantine consumers

We have carefully inspected how a byzantine consumer could influence unbonding on the provider; among inspected scenarios are, e.g. submitting multiple VSC maturity packets, or maturity packets with wrong unbonding ids, etc.

In the finding IF-ICS-05 we have shown a way for a byzantine consumer to delay the unbonding time on the provider chain to up 5 weeks by sitting on VSC maturity notifications until the very last moment. This is a strict upper bound: if a consumer delays sending the notifications any longer (or stops sending VSC maturity packets altogether), the provider's safety mechanism will remove the non-responsive consumer after the 5 weeks timeout. Thus, in the worst case, the unbondings could be delayed for 5 up to weeks.

Threat: Validators are harmed

In this section we analyze ways in which introducing Interchain Security to the Cosmos Hub could harm the validators. Again, we split the analysis into three parts: ICS without consumers, ICS with trustworthy consumers, and ICS with byzantine consumers.

No consumers

With Interchain Security, the `Staking` module will call into the hook `AfterUnbondingInitiated`, with the purpose of putting the unbonding on hold until the unbonding period has passed on all consumer chains. When there are no chains, however, the hook returns prematurely and does nothing. Thus, validators continue their operations uninterrupted. Furthermore, no slash packets can be received without existing consumer chains.

Trustworthy consumers

Trustworthiness of consumers assumes that the consumers send all their messages faithfully to the infraction that happened and on time. In our inspection of that case, we found that the validation proceeds as described in the ICS protocol specification.

Byzantine consumers

Byzantine consumers are the most dangerous threat to validators in the ICS protocol. The reason for it is that the protocol does not check evidence of misbehaviour provided by consumer chains. While the throttling mechanism prevents a takeover, it does not prevent a consumer chain from harming individual validators. Concretely, a byzantine consumer chain could tombstone an innocent validator. This scenario is examined in detail in the finding IF-ICS-01 and is already reworked in the ICS implementation.

We also inspected the throttling mechanism as a way to slow down an adversarial takeover of the validator set. We can confirm that the stated property of the mechanism (which makes sure that an attack would be slowed down) is correct. (For the clearest explanation of the property, please refer to the updated version.)

Findings

Severity	**Finding**
High	A byzantine consumer can tombstone, slash, or jail an innocent validator
Medium	Consecutive meter replenishments when the meter is negative
Medium	Panics on failure to send IBC packets
Low	Unbonding period on consumers is not enforced
Low	A consumer can delay unbonding period on the Hub to be up to 35 days
Low	A byzantine consumer might inflict Denial-Of-Service to other consumers
Informational	Suboptimal integration of changes into Cosmos SDK
Informational	Problems that harm code readability

A byzantine consumer can tombstone, slash, or jail an innocent validator

ID	IF-ICS-01
Severity	High
Impact	High
Exploitability	Medium
Type	Protocol assumption
Issue	link
Status Update	Resolved

Description

Per the Evidence Provision assumption, the provider will not check slash requests coming from consumers but will act upon them immediately. Thus, a byzantine consumer can report that a validator double signed a block, causing it to be slashed and tombstoned. Similarly, if a validator was reported for downtime, it will be slashed and jailed.

Problem Scenarios

Both described scenarios can be used to harm any individual validator.

Recommendation

We recommend checking the validity of an infraction report before acting upon it.

Status Update

Resolved. As a short-term solution, the provider will only jail validators for downtime infractions on consumers, there is no slashing or tombstoning (issue 689). In the future, the consumer will have to send evidence of an infraction to the provider (as planned in the Untrusted consumers milestone).

Consecutive meter replenishments when the meter is negative

ID	IF-ICS-02
Severity	Medium
Impact	Medium
Exploitability	Medium
Type	Implementation
Issue	link
Status Update	Fixed

Involved artifacts

- Documentation file throttle.md
- Implementation file throttle.go, containing the problem.
- Issue 684
- PR 687, which implements a fix for the problem.

Description

There is a bug in the implementation of the throttling mechanism. The protocol description states that

> The slash meter can go negative in value, and will do so when handling a single slash packet that jails a validator with significant voting power. In such a scenario, the slash meter may take multiple replenishment periods to once again reach a positive value, meaning no other slash packets may be handled for multiple replenishment periods.

However, the code does not implement this behavior correctly: If the meter has negative value, it will be replenished once. In the following block, it will be replenished again (because the meter was not full, so the `lastSlashMeterFullTime` variable will not get set (here). This enables the meter to get replenished every block until it finally becomes full and resets `lastSlashMeterFullTime`.

Problem Scenarios

This error undermines the guarantees of the throttling mechanism: it enables removing more validators from the active set than the protocol allows. A carefully crafted attack could aim at removing the largest-power validator just as the meter is very close to 0. Assuming that the removed validator has a fraction of total power `Vmax`, the meter is then replenished to its maximum in `Vmax/SlashMeterReplenishFraction + 1` *blocks*, when it should have been that same number of `SlashMeterReplenishPeriods`.

Recommendation

Implement a fix for the bug in the code by not resetting the time for when the meter could be replenished only when the meter was last time full. Instead, update that value each time the meter is replenished.

Status Update

Resolved. The problem is fixed by PR 687.

Panics on failure to send IBC packets

ID	IF-ICS-03
Severity	Medium
Impact	High
Exploitability	Low
Type	Implementation
Issue	link
Status Update	Unresolved

Involved artifacts

- x/ccv/provider/keeper/relay.go: SendVSCPacketsToChain()
- x/ccv/consumer/keeper/relay.go: SendPackets()

Description

In the implementation under audit, namely in the functions SendVSCPacketsToChain() on provider, and SendPackets() on consumer, the code panics if sending of IBC packets fails. As these functions are called from EndBlockers, the respective chain will halt on error. This has indeed happened in a testnet, when sending of an IBC packet failed because of an expired LightClient (see issue ICS#435). A partial fix has been introduced, in which the packets are first queued, and only then sent, with errors about light client expiration are logged, and the unsent packets remain queued. The resulting code looks similar on both provider and consumer; here is the relevant part of the provider code:

```
        // send packet over IBC
        err := utils.SendIBCPacket(...)

        if err != nil {
            if clienttypes.ErrClientNotActive.Is(err) {
                // IBC client is expired!
                // leave the packet data stored to be sent once the client is upgraded
                // the client cannot expire during iteration (in the middle of a block)
                k.Logger(ctx).Debug("IBC client is expired, cannot send VSC, leaving packet
↪  data stored:", "chainID", chainID, "vscid", data.ValsetUpdateId)
                return
            }
            // TODO do not panic if the send fails
            // https://github.com/cosmos/interchain-security/issues/649
            panic(fmt.Errorf("packet could not be sent over IBC: %w", err))
        }
```

As can be seen, upon receiving an error from sending an IBC packet, if the error is of one specific type (`ErrClientNotActive`), the function stops processing; *on any other returned error the function will panic, and thus halt the provider*. The problem with the presented approach is that it introduces an *implicit assumption* on the behavior of code that sends IBC packets, namely that the only error that it may return *under normal circumstances* is `ErrClientNotActive`. But even if this is true currently (though *normal circumstances* is hard to define), it can be seen that the function ibc-go/modules/core/04-channel/keeper/packet.go:SendPacket(): a) contains at least 10 different errors it may return, and b) is updated relatively frequently, so even if the above condition holds now, it may seize to hold upon the next IBC update.

Problem Scenarios

If, under some combination of parameters, or upon an IBC update, the IBC packet send returns an unexpected error, the provider or consumer chain will halt. The crux of the problem is in the immediate and harmful reaction to a problem that may have transient character.

Recommendation

In general, we recommend following the modular approach outlined in the issue ICS#627. Considering this specific case, the situation is much simplified by the fact that all infrastructure for delayed actions is already in place: IBC packets are queued both on the provider and on the consumer, and even if the packet can't be sent now, it can be resent later. Thus, we recommend the following:

- Remove the aforementioned panics on failure to send IBC packets
- Remove the special handling of the error `ErrClientNotActive`
- On *any error*, instead of panicking, issue an event, as well as a log message, describing the problem, and return from the function.
- Implement the monitoring solution that would watch for such events or log entries, and notify the human operators (e.g. validators) of the problem that ICS is unable to solve.

In that way, no immediate harmful actions will be taken wrt. either provider or consumer, and the problem will be delegated to humans for further actions; the ICS packets will then simply wait in the respective queues until the correcting actions are taken.

Unbonding period on consumers is not enforced

ID	IF-ICS-04
Severity	Low
Impact	Medium
Exploitability	Low
Type	Implementation
Issue	link
Status Update	Unresolved

Involved artifacts

- ibc/spec/app/ics-028-cross-chain-validation/methods.md
- interchain-security/docs/params.md
- interchain-security/x/ccv/provider/keeper/proposal.go: CreateConsumerClient()

Description

While the documentation says that unbonding period on consumer chains should be shorter than that on the provider chain, this is not enforced by the implementation.

In particular, in the file interchain-security/docs/params.md it is said that:

> The `ConsumerUnbondingPeriod` is set via the `ConsumerAdditionProposal` gov proposal to add a new consumer chain. Unbonding operations (such as undelegations) MUST wait for the unbonding period on every consumer to elapse. Therefore, for an improved user experience, the `ConsumerAdditionProposal` on every consumer chain SHOULD be smaller than `ProviderUnbondingPeriod`, i.e.,
>
> `ConsumerUnbondingPeriod = ProviderUnbondingPeriod - one day`

Also the ICS specification in the file ibc/spec/app/ics-028-cross-chain-validation/methods.md says, that the following postcondition should hold for the function `InitGenesis(gs: ConsumerGenesisState)`:

> `consumerUnbondingPeriod` is set to `ComputeConsumerUnbondingPeriod(gs.providerClientState.unbondingTime)`

with `ComputeConsumerUnbondingPeriod` being

```
function ComputeConsumerUnbondingPeriod(delta: Duration): Duration {
  if delta > 7*24*Hour {
    return delta - 24*Hour // one day less
  }
  else if delta >= 24*Hour {
        return delta - Hour // one hour less
    }
  return delta
}
```

But the actual implementation of the above, done in function CreateConsumerClient(), does nothing more than copying the unbonding period from the consumer chain governance proposal:

```
func (k Keeper) CreateConsumerClient(ctx sdk.Context, prop *types.ConsumerAdditionProposal)
↪  error {
    ...

    // Consumers always start out with the default unbonding period
    consumerUnbondingPeriod := prop.UnbondingPeriod
    // Create client state by getting template client from parameters and filling in zeroed
     ↪  fields from proposal.
    clientState := k.GetTemplateClient(ctx)
    clientState.ChainId = chainID
    clientState.LatestHeight = prop.InitialHeight

    trustPeriod, err := ccv.CalculateTrustPeriod(consumerUnbondingPeriod,
↪   k.GetTrustingPeriodFraction(ctx))
    if err != nil {
        return err
    }
    clientState.TrustingPeriod = trustPeriod
    clientState.UnbondingPeriod = consumerUnbondingPeriod

    consumerGen, validatorSetHash, err := k.MakeConsumerGenesis(ctx, prop)
```

Thus, nothing in the implementation really enforces the requirements stated in the specification, and the only enforcement mechanism for them is for voters on the governance proposal to check its parameters. While such a misconfiguration is unlikely to avoid detection, as consumer addition proposals will get lots of public attention, an automatic check should still be in place.

Problem Scenarios

A governance proposal could be passed with the unbonding period on the consumer chain larger than that on the provider chain. This will lead to all unbondings on the provider chain to be delayed for that period (e.g. to 5 weeks, instead of the current 3 weeks for Cosmos Hub).

Recommendation

Add enforcements mechanisms to the implementation, such that unbonding period on the consumer chains is smaller that that of the provider chain, as expected according to the specification.

A consumer can delay unbonding period on the Hub to be up to 35 days

ID	IF-ICS-05
Severity	Low
Impact	Medium
Exploitability	Low
Type	User experience
Status Update	Unresolved

Involved Artifacts

- function Complete Matured Unbondings

Description

With the addition of Interchain Security, the hub will allow unbonding only when all consumers confirm that their unbonding periods have passed. While a default unbonding period for a consumer is one day shorter than the Hub's 21 days, there is still a fail-safe mechanism that makes sure that an unresponsive consumer cannot delay the unbonding period for arbitrarily long - after 35 days from sending a VSC packet, a consumer which does not report back will be removed.

Problem Scenarios

A byzantine consumer can consistently wait for just under 35 days before sending a VSC maturity confirmation. As a consequence, the unbonding period of the Hub would turn into 35 days, worsening the overall user experience.

Recommendation

Add a monitoring mechanism to detect offending consumers as early as possible. In that way, repeated delays by the same consumer would be spotted early and a proposal to remove the consumer could be passed.

Status Update

Unresolved - the behaviour is a part of the protocol design.

A byzantine consumer might inflict Denial-Of-Service to other consumers

ID	IF-ICS-06
Severity	Low
Impact	Low
Exploitability	Medium
Type	User experience
Issue	link
Status Update	Unresolved

Involved artifacts

- the part of the code which panics when the queue is full
- issue 594, in which the problem is discussed
- issue 713, in which a solution is proposed

Description

As the provider is receiving packets from the consumer (both `OnRecvVSCMaturedPacket` and `OnRecvSlashPacket`), it enqueues them in a per-chain queue by calling `QueueThrottledPacketData`. That queue cannot grow larger than a parameter-specified limit, `MaxThrottledPackets`, whose default value is 100000. If the size of the queue exceeds the limit, the code will panic.

Problem Scenarios

This limit on the throttling queue size can be exploited by a byzantine consumer `C` to deny service to other consumers. Filling the queue with `C`'s packets only will make the provider unable to process any further packets from other consumers. Denying other consumers the opportunity to timely communicate with the provider could get them removed from Interchain Security entirely, because the provider will not receive relevant maturity notifications before the timeout.

Recommendation

There has already been a lively discussion on the topic in issue 594 before this audit. A likely solution will be to offload persisting of packets that cannot be processed immediately (the throttled ones) to consumers. We believe this is a good solution. Alternatively, an offending consumer (the one which is sending way too many packets) could be identified and removed.

Status Update

Unresolved. The problem is identified in issue 594 and a solution is proposed in issue 713.

Suboptimal integration of changes into Cosmos SDK

ID	IF-ICS-07
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status Update	Unresolved

Involved artifacts

- Changes to Cosmos SDK, branch v0.45.11-ics vs v0.45.11

Description

When integrating into the Cosmos SDK the changes necessary for Interchain Security, the obvious intention has been to minimally disrupt the already existing flow of control; see the overview of the changes in this document previously. At the same time, we believe that the changes have been done in a suboptimal way, thus unnecessary increasing the complexity of Cosmos SDK codebase. Below we describe the three cases where the changes could be done more efficiently.

Code duplication for completion of undelegations and reledelegations

Before the ICS-induced changes to Cosmos SDK, there have been only one way to complete undelegation and redelegation: when the corresponding entry matures (the unbonding period finishes). With the aforementioned changes, there appeared another way: when the entry matures locally on provider, but it is still "on hold" from consumers, it will complete later, when its reference counter reaches zero. For this new way of completing the entries the code that is doing the actual work has been more or less verbatim copied from the original variants:

- the new code of unbondingDelegationEntryCanComplete() copies to a large extent the code of CompleteUnbonding()
- the new code of redelegationEntryCanComplete() is to a large extent the duplicate of the code of CompleteRedelegation().

Code duplication in utility functions for handling unbonding operations

The newly introduced file x/staking/keeper/unbonding.go contains multiple utility functions for handling undelegations, redelegations, and validator unbondings. Many of these functions differ only in small details (e.g. in the value of the store key used), and thus can be collapsed into a single function with different arguments provided at call sites.

Problem Scenarios

Code duplication, as described above, while not being an immediate correctness problem or a threat, represents an architectural problem for Cosmos SDK down the road, especially when combined with other similar changes:

- The resulting code is bloated, increasing the maintenance cost (more tests needed, changes become more costly, etc.). Moreover, each new change has the potential to multiply with the previous ones, thus leading to an exponential growth of the codebase.
- Code duplication constitutes the potential for subtle bugs later, when e.g. the code that should behave in the same way, is updated only in one of the several places of duplication.

Recommendation

Not an immediately pressing concern, but in the mid term we recommend to refactor the code, and abstract the functions with duplicate or very similar functionality.

Problems that harm code readability

ID	IF-ICS-08
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Issue	link
Status Update	Unresolved

Involved artifacts

- obsolete comment 1
- obsolete comment 2
- dead code
- a function with a named return, but with explicit returns
- naked returns in a complex function (potentially confusing)
- issue 671
- issue 688

Description

There are a couple of places in the code which harm readability. In particular, there are instances of obsolete and confusing comments, dead code, and inappropriately used naked returns.

The comment before setting the order of EndBlockers is obsolete and confusing: in the audited version of the code, a consumer client is actually created in the `BeginBlock`, not in the `gov` module. Similarly, there is a comment before the function `DistributeRewardsInternally` that is not true anymore.

The function `NewHandler` is not used at all. It is supposed to call the function `AssignConsumerKey` upon receiving the corresponding message, but instead, the same functionality is accomplished with `tx.pb.go` and `codec.go`.

Finally, there are some functions using naked returns in a confusing way. The function `StopConsumerChain` names its return `err` of type `error`, but always returns explicitly. We recommend removing the named return value.

The function `RemoveConsumerFromUnbondingOp` uses a boolean naked return. Given that the function is not trivial, we recommend being explicit about the return value.

Status Update

Unresolved. Part of the finding are scheduled to be resolved in one of the next releases (see issue 671 and issue 688).

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for $< 3\%$; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

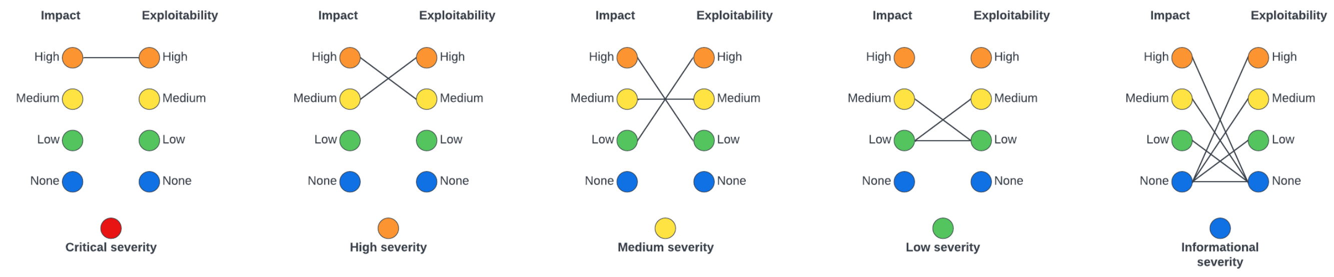

Figure 6: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary