Security Audit Report

Stride StakeIBC and ICACallbacks Modules: Source Code Analysis

2022/11/30

Last revised 2022/12/08

Authors: Darko Deuric, Andrey Kuprianov, Marko Juric

Contents

Audit overview **5**
The Project . . . 5
Scope of this report . . . 5
Conducted work . . . 5
Timeline . . . 5
Conclusions . . . 6
Further Increasing Confidence . . . 6

System Overview **7**
Data flow diagrams . . . 7
Deposit & Liquid Staking . . . 8
Staking & Reinvesment . . . 8
Unbonding . . . 10
IBC/ICA function call hierarchies . . . 10

Methodology **12**
Vulnerability classification . . . 12
Impact Score . . . 12
Exploitability Score . . . 12
Severity Score . . . 13

Audit Dashboard **15**

Findings **16**

Unsafe usage of native arithmetic may lead to catastrophic failures **17**
Involved artifacts . . . 17
Description . . . 17
Problem Scenarios . . . 19
Recommendation . . . 19

Unbonding is compromised if the overflow value is greater than zero **20**
Involved artifacts . . . 20
Description . . . 20
Problem Scenarios . . . 21
Recommendation . . . 21

One chain redemption out-of-bounds may halt all chains **23**
Involved artifacts . . . 23
Description . . . 23
Problem Scenarios . . . 23
Recommendation . . . 24

IBC/tokens could be lost during IBC transfer to Delegation ICA **26**
Involved artifacts . . . 26
Description . . . 26
Problem Scenarios . . . 27
Recommendation . . . 28

Users may not be able to redeem stake **29**
Involved artifacts . . . 29
Description . . . 29

Problem Scenarios . . . 29
Recommendation . . . 29

StakeExistingDepositsOnHostZones could be a bottleneck in case of many host zones **30**
Involved artifacts . . . 30
Description . . . 30
Problem Scenarios . . . 30
Recommendation . . . 30

Failure to send IBC packets may lead to user funds freeze **31**
Involved artifacts . . . 31
Description . . . 31
Problem Scenarios . . . 31
Recommendation . . . 32

Consider isolating host zone operations **33**
Involved artifacts . . . 33
Description . . . 33
Problem Scenarios . . . 33
Recommendation . . . 35

GetHostZoneUnbondingMsgs need to be refactored **36**
Involved artifacts . . . 36
Description . . . 36
Problem Scenarios . . . 37
Recommendation . . . 37

Variable err not assigned but used inside if condition **39**
Involved artifacts . . . 39
Description . . . 39
Problem Scenarios . . . 39
Recommendation . . . 39

Error handling should be reviewed **40**
Involved artifacts . . . 40
Description . . . 40
Problem Scenarios . . . 41
Recommendation . . . 41

Different conditions for max number of validators to rebalance **42**
Involved artifacts . . . 42
Description . . . 42
Problem Scenarios . . . 42
Recommendation . . . 42

Redemption rate limits are hardcoded **44**
Involved artifacts . . . 44
Description . . . 44
Problem Scenarios . . . 44
Recommendation . . . 44

Documentation is not updated (including Linux support) **45**
Involved artifacts . . . 45
Description . . . 45
Problem Scenarios . . . 46
Recommendation . . . 46

Coding style recommendations **47**
Involved artifacts . . . 47
Description . . . 47
Recommendation . . . 48

Stride lacks various test cases **49**
Involved artifacts . . . 49
Description . . . 49
Recommendation . . . 49

Various kinds of observations **50**
Description . . . 50
Problem Scenarios . . . 50
Recommendation . . . 51

Audit overview

The Project

In October 2022, Stride development team engaged Informal Systems to conduct a security audit over the documentation and the current state of the Stride modules named: *stakeibc* and *icacallbacks*.

Stakeibc module is the main module of Stride blockchain responsible for all of the interchain (un)staking and reinvestment logic as well as for minting & burning derivative assets (stTokens). It owns 4 Interchain Accounts (ICAs) (Delegate, Withdraw, Redemption, and Fee) on each host stride interacts with. It issues all ICA messages like MsgDelegate, MsgUndelegate, MsgBankSend, IBC transfers. Staking yield accrues to stTokens. Rewards are socialized across all depositors. *Icacallbacks* is an auxiliary model that allows to store callbacks to be called on IBC packet acknowledgements; the stored callbacks constitute the indivisible part of the *stakeibc* module logic.

Scope of this report

The agreed-upon work plan consisted of the following tasks:

$Task1$. **x/stakingibc** - edge cases around ICA (whether all edge cases are handled, like packet acks and failures

$Task2$. **x/stakingibc** - the situation when transfer to the Delegation account fails

$Task3$. **x/stakingibc** - the correctness of the core functions (MsgLiquidStake, MsgRedeemStake, MsgClaimStake)

$Task4$. **x/stakingibc** - whether the icacallbacks in stakeibc are executing as expected

$Task5$. **x/icacallbacks** - fully audit the module and check for correctness

This report covers the above tasks that were conducted from October 12 through November 30 by Informal Systems by the following personnel:

- Darko Deuric
- Andrey Kuprianov
- Marko Juric

Conducted work

Starting with October 12, The Informal Systems team audited the existing documentation and the code. On the same date, the Stride development team representative gave us an hour presentation about the Stride multichain dataflow, with IBC/ICA focus and we define the scope of this audit. The additional details including which tag to use (v2.0.3) to perform an audit for were agreed on at this meeting.

The team reviewed all the existing specifications and technical diagrams for the x/stakeibc and x/icacallbacks deep dive. Our team performed the high-quality line-by-line manual code review of the two modules previously mentioned with the main focus on IBC/ICA correctness together with the callbacks mechanism used for updating the state on Stride chain on packet acknowledgment. Over the shared Slack channel, we discussed setting the local test environment, how the validators are being updated, and general work done during the online weekly sync meetings.

Timeline

- 10/12/2022: kickoff meeting with Stride team (Aidan and Riley talked about general data flow in Stride blockchain)
- 10/19/2022: 1st sync meeting (general code observations, first critical finding discussed - panic related to redemption rate limits)
- 10/27/2022: 2nd sync meeting (walkthrough findings related to liquid staking, delegation and reinvestment, Linux installation problems presented, Aidan answered a few questions)

- 10/28/2022: collaboration repo created and 1st finding added by Andrey
- 11/3/2022: 3rd sync meeting (additional findings and observations reported, mainly about unbonding process)
- 11/4/2022: Stride team provided dockernet branch for Linux
- 11/9/2022: 4th sync meeting (majority of findings added to collaboration repo, a short discussion about validators)
- 11/30/2022: meeting Informal/Stride to gain feedback on the audit report

Conclusions

We found that the x/stakeibc and x/icacallbacks modules design and security models in general are well thought out. The amount of existing unit tests and existing development practices are on a very high level although integration tests could be improved because the only integration test that has been written describes the happy-path so an illusion of security that does not really exist in real life can be provided. There are plenty of potential error messages defined inside stakeibc module so at least some of them should be checked with integration tests.

Despite the general high quality, we found some details that should be addressed in order to raise the quality of code and existing specification. **1 Critical severity** and **4 High Severity** issues were found during this audit; the rest were marked Medium, Low or Informational severity. Until this time, one issue was addressed.

Further Increasing Confidence

The scope of this audit was limited to manual code review and manual analysis and reconstruction of the protocols. To further increase confidence in the protocol and the implementation, we recommend following up with more rigorous formal measures, including automated model checking and model-based adversarial testing. Our experience shows that incorporating test suites driven by TLA+ models that can lead the implementation into suspected edge cases and error scenarios enable discovery of issues that are unlikely to be identified through manual review.

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

System Overview

In Stride, staking occurs every 6 hours and it goes through 3 epochs:

- **epoch n**: New deposit record (DR) which tracks all deposits in a given epoch for a given host zone is created with 0 tokens. LiquidStake is called then and stTokens are minted, but the actual staking of the user's tokens is addressed in epoch n+2.
- **epoch n+1**: All tokens on DR from epoch n are IBC transferred from Stride's module account to Delegation ICA. Whenever a change to delegation happens all of the rewards are withdrawn (to Withdraw ICA).
- **epoch n+2**: Tokens on the DR are staked (by weight) across all (30 at the moment) Stride validators.

Reinvestment executes automatically and the rewards are auto-compounded on every epoch (6h). 90% of the rewards are being sent to Delegation ICA (reinvestment) and 10% to the Fee ICA (the only place where Stride charges the fees)

- **epoch n**: Queries Interchain Query (ICQ) to check balances of Withdraw ICA and creates a new record for those tokens.
- **epoch n+1**: Transfers tokens to Delegation ICA from the Withdraw ICA.
- **epoch n+2**: Stakes the tokens.

Unstaking executes every day. Only 7 concurrent unbondings are allowed (a constraint on Cosmos) on host zones for a delegator and validator pair.

- **epoch n**: EpochUnbondingRecord is created. It stores many HostZoneUnbondings (HZU), with one HZU per host zone (e.g. for CosmosHub we have 1 HZU per epoch). When the user sends 1stAtom to Stride (Stride now custodies this 1 stAtom) and specifies an address on the Cosmos Hub that the tokens should be sent, HZU is updated and UserRedemptionRecord (URR) is created (claim on user's tokens that they can trigger later once the tokens have unbonded).
- **epoch n+m** (m = unbondingPeriodOnHostZone / 7 + 1): For CosmosHub it happens every 4 days (unbPeriod = 21 days). MsgUndelegate is triggered and all of the pulled unbonding tokens are undelegated (MsgUndelegate ICAs are triggered across the validators Stride has delegated to). HZUs are updated with unbonding time.
- **epoch n+unbonding time**: Tokens are transferred to Redemption ICA account. The URR is updated so that the tokens are claimable and anyone can transfer tokens to the already specified address which is stored on URR. So this is an ICA call that transfers tokens back to the end user's account.

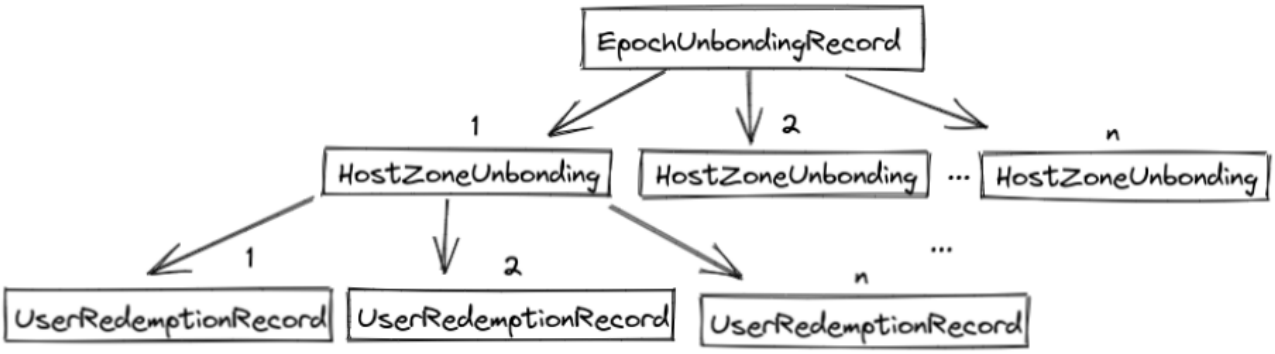

Figure 1: Epoch unbonding record

Data flow diagrams

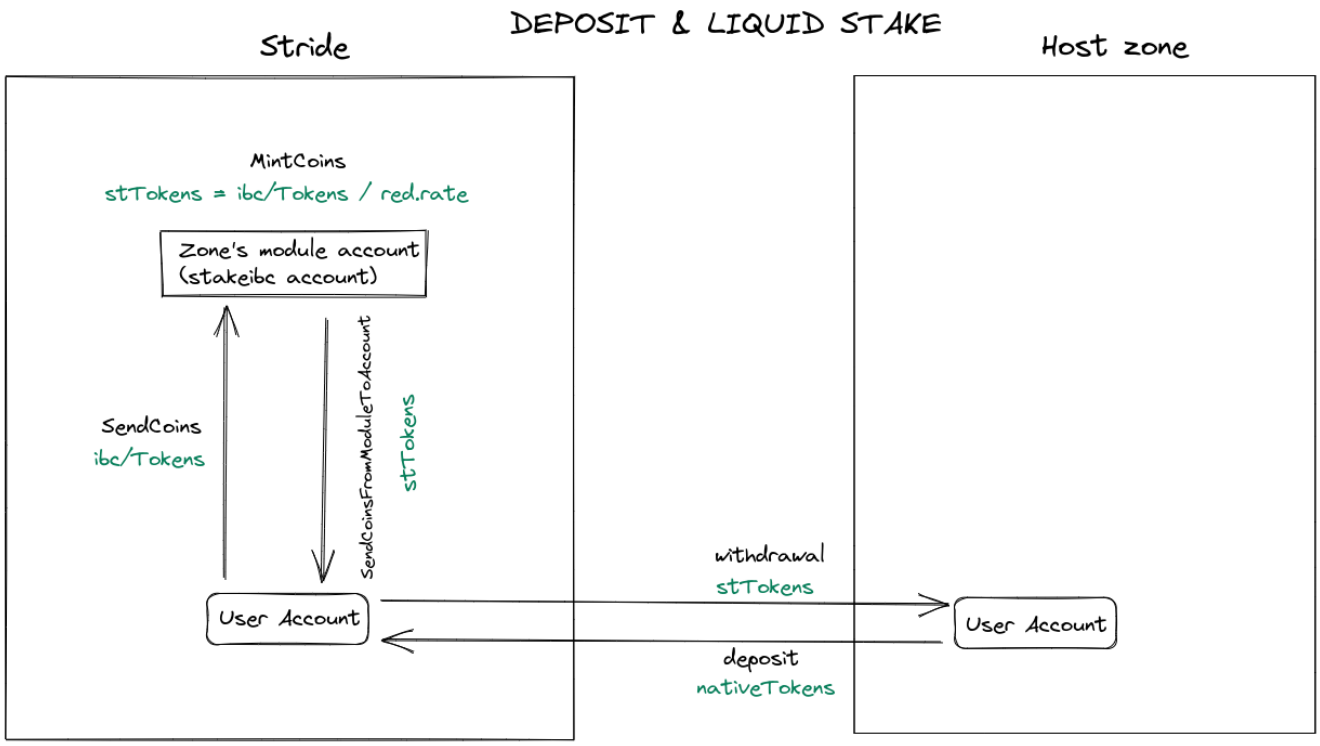

Figure 2: Deposit & Liquid Staking

Deposit & Liquid Staking

Withdrawal and deposit belong to regular bank transfers (outside of Stride). After transferring native tokens to Stride, liquid staking can be processed and it includes:

1. Sending IBC/Tokens to stakeibc account
2. Minting stTokens to stakeibc account
3. Sending stTokens from stakeibc account to user account on Stride

Staking & Reinvesment

Staking and reinvestment steps:

1. Sweeps the deposit record (DR) marked `TRANSFER_QUEUE` from previous epochs. Under the hub, it constructs IBC `MsgTransfer` with 30min timeout. `TransferCallback` is also created which is been called `OnAcknowledgementPacket` or `OnTimeoutPacket`. In the case of nill ack or ack_error DR's status is set back to `TRANSFER_QUEUE` otherwise it becomes a candidate for delegation with `DELEGATION_QUEUE` flag.
2. Delegates DRs with status `DELEGATION_QUEUE`. It creates a set of `MsgDelegate` msgs (delegation to every validator from that host zone whose relative amount is positive). Each validator gets `targetAmount=valWeight*depRecordAmount / totalValWeight`. Also, `DelegateCallback` is defined. In the case of the happy ibc path, the zone's staked balance is increased by a delegated amount to each validator whose `delegationAmt` is updated and finally DR is removed.
3. Rewards are automatically sent to the Withdrawal ICA.
4. `WithdrawalBalanceCallback` executes ICA SendTx with `MsgSend` to `Delegation ICA` (90% reward) and `Fee ICA` (10% reward). It also adds `ReinvestCallback` which triggers as previous ones, and in the happy path, it creates a new DR with `DELEGATION_QUEUE` status (using `WITHDRAWAL_ICA` source this time, not `STRIDE` source) while the sad path is ignored.

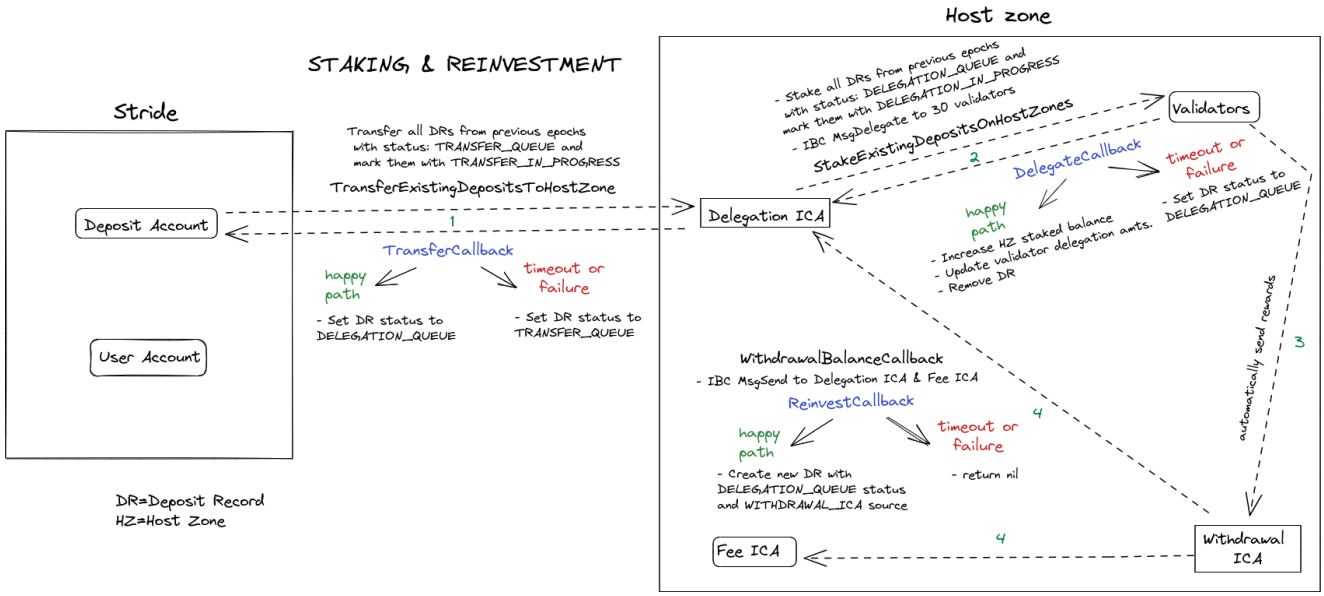

Figure 3: Staking & Reinvesment

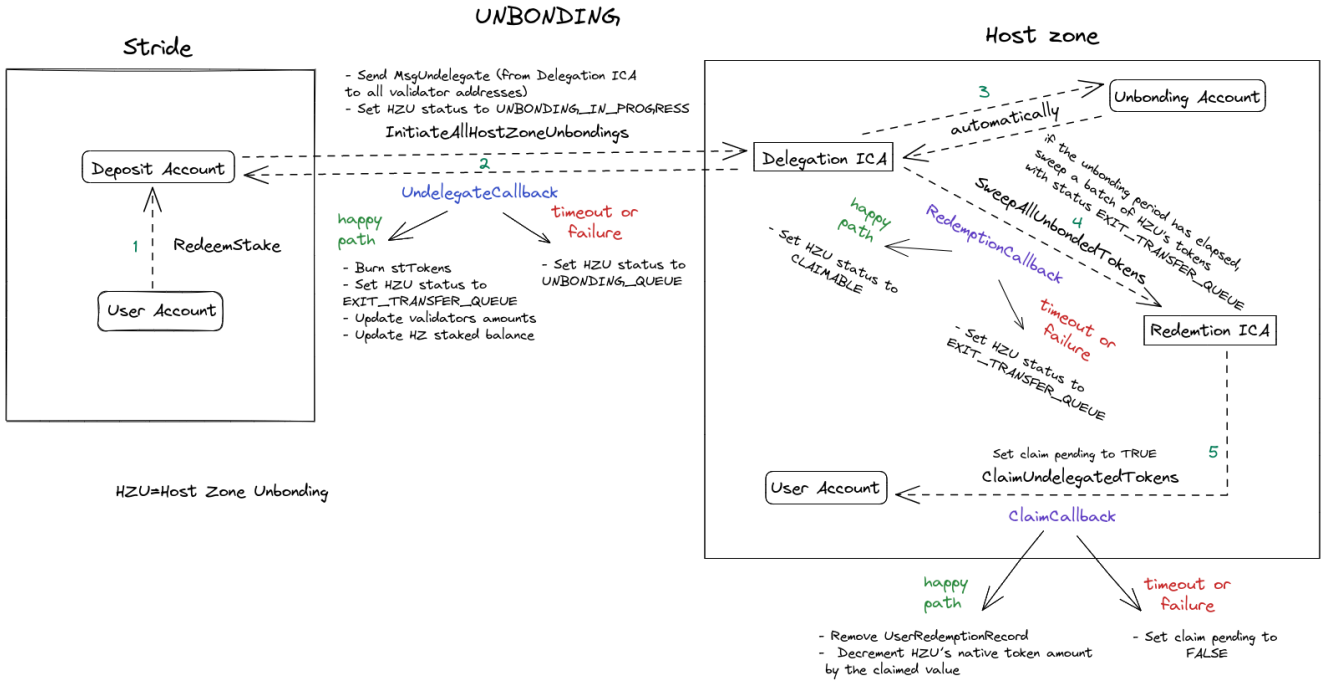

Figure 4: Unbonding

Unbonding

Unbonding goes as follows:

1. User sends `RedeemStake` msg which creates `UserRedemptionRecord` (URR), calculates the amount of native tokens to get back: `nativeAmount=stAmount*redemptionRate`, updates `HostZoneUnbonding` (HZU) record and sends `numStAtoms` to module's account where they will eventually be burned after unbonding.
2. On every epoch (a day) initiates unbondings for all host zones with ICA `SendTx` containing `MsgUndelegate` (from `Delegation ICA` to all validators) and sets HZU status to `UNBONDING_IN_PROGRESS` (note: HZU is created at `RegisterHostZone` with initial status set to `UNBONDING_QUEUE`). If ICA `SendTx` fails, `UndelegateCallback` will roll back the HZU status to `UNBONDING_QUEUE`, otherwise it burns escrowed `stTokens`, updates: HZU status to `EXIT_TRANSFER_QUEUE`, HostZone staked balance and validator amounts.
3. The funds are in escrow for the unbonding period and then get automatically transferred to back to the `Delegation ICA`.
4. Once per epoch if the unbonding period has elapsed, all HZU's tokens with `EXIT_TRANSFER_QUEUE` are swept as a batch to `Redemption ICA`. If everything goes well, HZU status is set to `CLAIMABLE`, otherwise it's reset to `EXIT_TRANSFER_QUEUE` during `RedemptionCallback` execution.
5. Users can claim their unbonded tokes via `ClaimUndelegatedToken` which also sets claim pending to TRUE (to avoid double claims). ICA SendTx is constructed with MsgSend inside (from Redemption ICA to user address on host) with 10min timeout. If no ack errors, URR is removed and HZU's native token amount is decremented by the claimed value. Else, if timeot or ack failure, claim pending is set to false.

IBC/ICA function call hierarchies

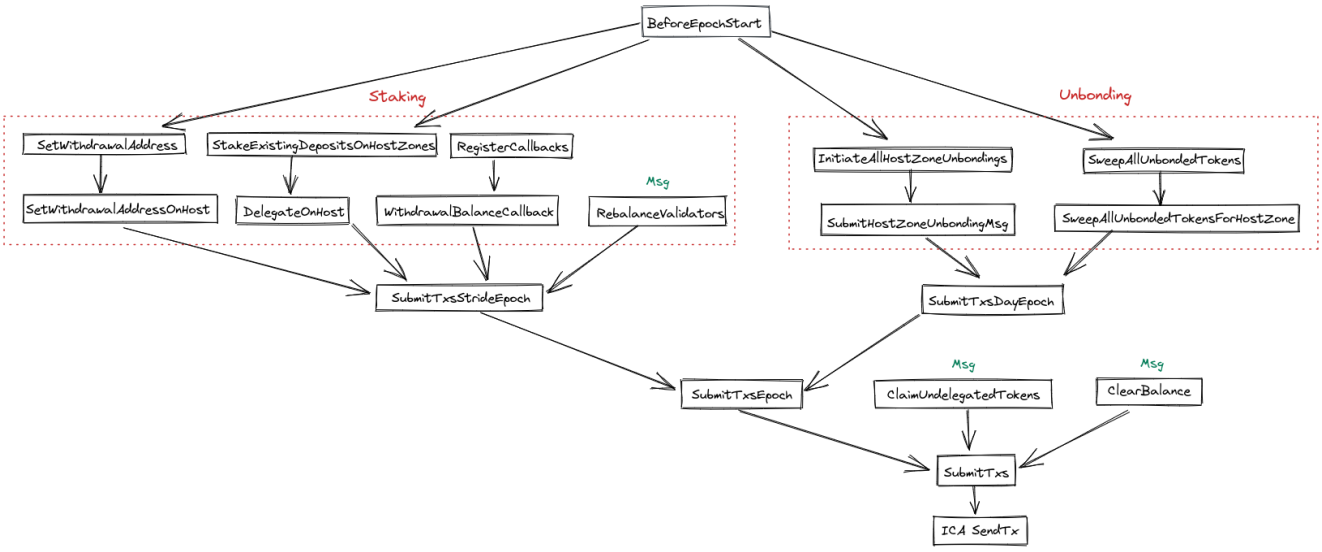

Figure 5: ICA SendTx call hierarchy

It can be seen that many paths lead to ICA's `SendTx`, including both, staking and unstaking parts of the picture while all paths (except direct Msgs) have the same root - `BeforeEpochStart` function.

There are only two scenarios for ICA's `RegisterInterchainAccount` to be called. The first one is when registering the host zone, 4 ICAs are created. The second one, in the case of timeout, if a channel closes, the controller chain must be able to regain access to registered interchain accounts by simply opening a new channel which is done through `RestoreInterchainAccount`.

IBC `Transfer` is called at one place only from record's module when sweeping existing deposits from Stride to the Host Zones.

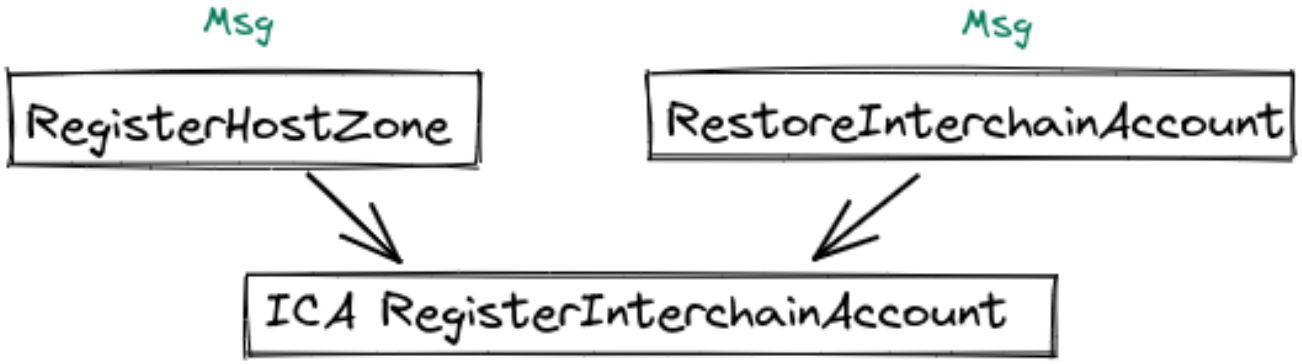

Figure 6: ICA RegisterInterchainAccount call hierarchy

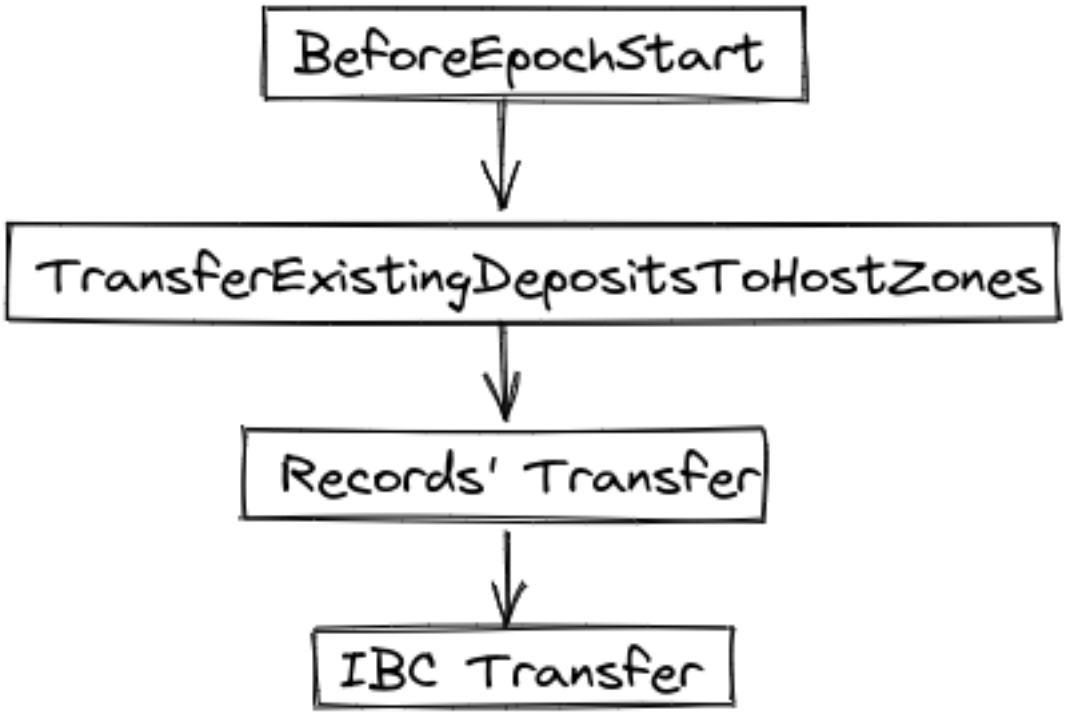

Figure 7: IBC transfer call hierarchy

Methodology

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.

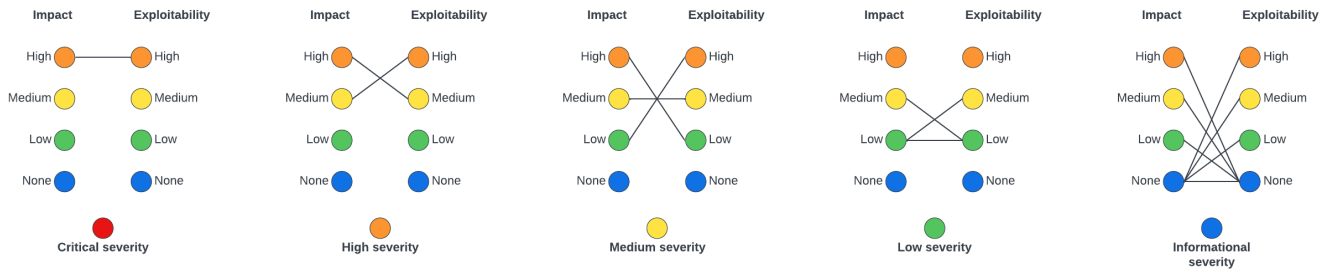

Figure 8: Severity classification

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

SeverityScore	Examples
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

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Golang
- **Artifacts**
  - Stride-Labs/stride/x/stakeibc @ `<v2.0.3>`
  - Stride-Labs/stride/x/icacallbacks @ `<v2.0.3>`

Engagement Summary

- **Dates**: 10/12/2022 to 11/30/2022
- **Method**: Manual code review
- **Employees Engaged**: 3

Severity Summary

Finding Severity	#
Critical	1
High	4
Medium	2
Low	0
Informational	10
Total	**17**

Findings

Severity	Finding
Critical	Unsafe usage of native arithmetic may lead to catastrophic failures
High	Unbonding is compromised if the overflow value is greater than zero
High	One chain redemption out-of-bounds may halt all chains
High	IBC/tokens could be lost during IBC transfer to Delegation ICA
High	Users may not be able to redeem stake
Medium	StakeExistingDepositsOnHostZones could be a bottleneck in case of many host zones
Medium	Failure to send IBC packets may lead to user funds freeze
Informational	Consider isolating host zone operations
Informational	GetHostZoneUnbondingMsgs need to be refactored
Informational	Variable err not assigned but used inside if condition
Informational	Error handling should be reviewed
Informational	Different conditions for max number of validators to rebalance
Informational	Redemption rate limits are hardcoded
Informational	Documentation is not updated (including Linux support)
Informational	Coding style recommendations
Informational	Stride lacks various test cases
Informational	Various kinds of observations

Unsafe usage of native arithmetic may lead to catastrophic failures

ID	IF-STRIDE-STAKEIBC-ARITHMETIC
Severity	Critical
Impact	High
Exploitability	High
Type	Implementation
Status	Unresolved

Involved artifacts

- stakeibc/keeper/msg_server_liquid_stake.go
- stakeibc/keeper/validator_selection.go
- a multitude of other files partially listed below.

Description

In multiple places throughout the `StakeIBC` module native Go numbers of types `int64`, `uint64`, `float64` are used. In general, usage of native types, especially with such low bit width should be avoided by any means due to the following reasons:

- native types have a limited (and in case of 64 bits also small!) range of numbers they can represent: see Go numeric types;
- **arithmetic operations on integer native types overflow/underflow without any errors or warnings**, they simply wrap around the range. Thus:
  - `int64` on arithmetic overflow (above `9223372036854775807`) starts again from the smallest negative number (`-9223372036854775808`); and on underflow the other way around;
  - `uint64` on arithmetic overflow (above `18446744073709551615`) starts again from `0`; while on underflow (when the result falls below `0`) – counts back from largest number (`18446744073709551615`).
- arithmetic operations on `float64` is less likely to overflow or underflow, as the representable range is quite large (between ~ `-1.7e+308` to `+1.7e+308`), but `float64` looses precision quite quickly, it can represent up to around 15 significant decimal digits (52 bits). **Precision loss happens also without any errors or warnings.**

Below we show two places in StakeIBC where unsafe arithmetic operations are employed.

Possible overflow while updating deposit records in `msg_server_liquid_stake`

In `stakeibc/keeper/msg_server_liquid_stake.go` the following fragment is present:

```
msgAmt, err := cast.ToInt64E(msg.Amount)
if err != nil {
    k.Logger(ctx).Error("failed to convert msg.Amount to int64")
    return nil, sdkerrors.Wrapf(err, "failed to convert msg.Amount to int64")
}
depositRecord.Amount += msgAmt
k.RecordsKeeper.SetDepositRecord(ctx, *depositRecord)
```

Here `msg.Amount` as defined in `MsgLiquidStake` has type `uint64`, which is then cast to `msgAmt` of type `int64`; the latter is added to `depositRecord.Amount`, also of type `int64`. Moreover, `msg.Amount` comes from the user supplied

transaction, and thus can be anything, provided the constraints in the code above these lines are satisfied (i.e. the user has the `Amount` of coins). `depositRecord.Amount` sums up all user liquid stake transactions within an epoch, and this can be a large number.

Whenever the next total amount accumulated in a deposit record exceeds maximum value for `int64` (`9223372036854775807`), the operation `depositRecord.Amount += msgAmt` will silently overflow, counting anything that goes above maximum from `-9223372036854775808`; thus `depositRecord.Amount` will be assigned a large negative value. What is likely to happen after that is when the deposit record is transferred to the host zone in function TransferExistingDepositsToHostZones(), in particular here, the deposit record will be wiped out as the record with negative balance, and all user funds will be permanently lost:

```
if depositRecord.Amount <= 0 && depositRecord.DepositEpochNumber < epochNumber {
    k.Logger(ctx).Info("[TransferExistingDepositsToHostZones] Empty deposit record (ID: %s)!
↪  Removing.", depositRecord.Id)
    k.RecordsKeeper.RemoveDepositRecord(ctx, depositRecord.Id)
    continue
}
```

Possible overflow while distributing deposits to validators in `validator_selection`

In the function GetTargetValAmtsForHostZone(), which calculates the amounts to be distributed to validators on a particular host zone, we find the following fragment:

```
    for i, validator := range hostZone.Validators {
        if i == len(hostZone.Validators)-1 {
            // for the last element, we need to make sure that the allocatedAmt is equal to the
            ↪  finalDelegation
            targetAmount[validator.GetAddress()] = finalDelegation - allocatedAmt
        } else {
            delegateAmt, err := cast.ToUint64E(float64(validator.Weight*finalDelegation) /
↪  float64(totalWeight))
            if err != nil {
                k.Logger(ctx).Error(fmt.Sprintf("Error getting target weights for host zone
↪  %s", hostZone.ChainId))
                return nil, err
            }
            allocatedAmt += delegateAmt
            targetAmount[validator.GetAddress()] = delegateAmt
        }
    }
```

A particularly interesting line is this:

```
delegateAmt, err := cast.ToUint64E(float64(validator.Weight*finalDelegation) /
↪  float64(totalWeight))
```

Both `validator.Weight` and `finalDelegation` are of type `uint64`; the latter represents the total number of tokens to delegate, while the former – the weight of a particular validator. We have asked Stride for data on possible validator weights, and have received a table, where the maximal validator weight was `131604`. The multiplication between those two numbers is done within the type `uint64`, before being cast to `float64`. Thus, this intermediate operation is especially likely to overflow; and as the `Weight` multiplier is sufficiently large, the result can be anywhere in the range of `uint64`. As further computations are based on this intermediate result, this opens a wide range of possibilities for an attacker. One such possibility is to intentionally delegate such amount of tokens that:

- the delegated amount will overflow for the largest validator in such a way that the resulting `delegateAmt` for that validator would be close to `0`;
- the delegated amounts for all other validators, besides the last one will stay as intended;
- the last validator will get delegated all tokens intended to be delegated for the largest validator, thus substantially changing the validator power distribution.

Other potential problematic locations

A non-exhaustive list of other places in the code where unsafe arithmetic operations are performed is listed below:

- `stakeibc/keeper/hooks.go`: L240, L263, L278
- `stakeibc/keeper/icacallbacks_delegate.go`: L87
- `stakeibc/keeper/icacallbacks_rebalance.go`: L80, L85
- `stakeibc/keeper/icacallbacks_undelegate.go`: L128, L190
- stakeibc/keeper/validator_selection.go: multiple places
- stakeibc/keeper/unbonding_records.go: multiple places
- stakeibc/keeper/callbacks.go: multiple places
- stakeibc/keeper/msg_server_rebalance_validators.go: multiple places

We recommend paying particular attention to the last file in the above list: besides the multitude and complexity of possibly overflowing/underflowing arithmetic operations, the overall complexity of the implemented algorithm seems too high; we would recommend refactoring and simplifying it. We also recommend avoiding the usage of Cosmos SDK types/Dec datatype, due to suspicions we have in the correctness of its implementation.

Problem Scenarios

When an overflow/underflow/precision loss during operations on native numbers happens, the control flow will proceed undisturbed, using wrong data; this may lead to such catastrophic consequences as:

- halting of the chain;
- permanent loss of user funds;
- withdrawal of funds by unauthorized actors;
- unauthorized and disproportional changes to validator power.

We rate the impact of this finding as **High**, because of numerous and disastrous consequences this vulnerability may have. We rate the exploitability of this finding as **High** as well, because of the multitude of possible places where this vulnerability may be triggered. Thus, the vulnerability receives the overall severity score of **Critical**, and we recommend to address it as soon as possible.

Recommendation

This class of bugs can be avoided completely via usage of appropriate data types. We recommend the following:

- for *storage/transfer representation* (e.g. Cosmos SDK storage or Protobuf), select fixed bit width data types (e.g. `uint64`) only when you are absolutely sure that the semantic values they represent can never exceed the representable maximum over the whole lifespan of the system; otherwise use a variable length encoding, represented e.g. via `string`.
- for *internal representation*, in which all arithmetic operations are performed, employ a datatype with large bit width (e.g. 256 bits), and automatic checking of under- and overflows, a good candidate for that in the context of Cosmos SDK is math/Int.
- always perform "*safety border control*": transform from transfer/storage representation into internal representation as soon as data enters the system, and in the opposite direction immediately before data leaves the system.

A separate problem arises wrt. handing of overflows/underflows in "safe" datatypes such as math/Int. While such datatypes do check for error conditions, the only way for them to signal an error is via `panic`-ing. As most of Stride code executes in the context of `BeginBlocker`s and `EndBlocker`s (either directly, or indirectly, via epochs), panicking would cause similarly severe consequences (halting the chain). To avoid such consequences we recommend employing the Defer/Panic/Recover idiom at the top level of Stride codebase.

Unbonding is compromised if the overflow value is greater than zero

ID	IF-STRIDE-STAKEIBC-UNBONDING_COMPROMISED
Severity	High
Impact	Medium
Exploitability	High
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/unbonding_records.go

Description

If there is an overflow of unbonding tokens which happens:

```
if valUnbondAmt > currentAmtStaked { // if we don't have enough assets to unbond
            overflowAmt += valUnbondAmt - currentAmtStaked
            valUnbondAmt = currentAmtStaked
        }
```

and the overflow is not distributed to validators:

```
if overflowAmt > 0 { // what?? we still can't cover the overflow? something is very wrong
        errMsg := fmt.Sprintf("Could not unbond %d on Host Zone %s, unable to balance the
↪  unbond amount across validators",
            totalAmtToUnbond, hostZone.ChainId)
        k.Logger(ctx).Error(errMsg)
        return nil, 0, nil, nil, sdkerrors.Wrap(sdkerrors.ErrNotFound, errMsg)
    }
```

error is returned and `SubmitHostZoneUnbondingMsg` is not called. `overflowAmt` directly depends on `totalAmtToUnbond` which is the sum of all host zone's native token amounts for unbonding:

```
// mark the epoch unbonding record for processing if it's bonded and the host zone unbonding
↪  has an amount g.t. zero
if hostZoneRecord.Status == recordstypes.HostZoneUnbonding_UNBONDING_QUEUE &&
↪  hostZoneRecord.NativeTokenAmount > 0 {
    totalAmtToUnbond += hostZoneRecord.NativeTokenAmount
    epochUnbondingRecordIds = append(epochUnbondingRecordIds, epochUnbonding.EpochNumber)
    k.Logger(ctx).Info(fmt.Sprintf("[SendHostZoneUnbondings] Sending unbondings, host zone: %s,
↪  epochUnbonding: %v", hostZone.ChainId, epochUnbonding))

    }
```

On the other hand, `hostZoneRecord.NativeTokenAmount` represents the sum of unbonding tokens for each host zone that's updated on every RedeemStake call:

```
hostZoneUnbonding.NativeTokenAmount += nativeAmount.Uint64()
```

where:

```
nativeAmount := sdk.NewDec(amt).Mul(hostZone.RedemptionRate).RoundInt()
```

and `amt` originates from the user's msg.Amount:

```
amt, err := cast.ToInt64E(msg.Amount)
```

Problem Scenarios

The condition below could be "attacked" by enforcing `overflowAmt` to be greater than zero.

```
if overflowAmt > 0 { // what?? we still can't cover the overflow? something is very wrong
        errMsg := fmt.Sprintf("Could not unbond %d on Host Zone %s, unable to balance the
↪  unbond amount across validators",
            totalAmtToUnbond, hostZone.ChainId)
        k.Logger(ctx).Error(errMsg)
        return nil, 0, nil, nil, sdkerrors.Wrap(sdkerrors.ErrNotFound, errMsg)
    }
```

Although there is a check inside RedeemStake:

```
if msg.Amount > hostZone.StakedBal {
        return nil, sdkerrors.Wrapf(types.ErrInvalidAmount, "cannot unstake an amount g.t.
        ↪  staked balance on host zone: %d", msg.Amount)
    }
```

that should really be:

```
if nativeAmount > hostZone.StakedBal {
        return nil, sdkerrors.Wrapf(types.ErrInvalidAmount, "cannot unstake an amount g.t.
        ↪  staked balance on host zone: %d", msg.Amount)
    }
```

even with that, one or many users could (many times) provide `nativeAmount` slightly less than `hostZone.StakedBal`, so `hostZoneUnbonding.NativeTokenAmount` may become very large:

```
hostZoneUnbonding.NativeTokenAmount += nativeAmount.Uint64()
```

and exceed the total host zone's staked value. That way unbonding would be compromised for one or many host zones because following the chain from `hostZoneUnbonding.NativeTokenAmount` to `overflowAmt`, the latter may be greater than zero thus preventing a certain amount of users from unbonding their tokens. Our feeling is that a relatively small number of users may slow down/block the unbonding process (by sending large enough redeem amounts many times). But probably they will not be able to block the unbonding process completely, so the exploitability of this finding is rated as Medium. We rate the impact of this finding also as Medium, because it will not halt the chain but may have temporary denial of service / unexpected delays in processing user requests (unbonding). Thus, we assign this finding the overall severity score of Medium.

Recommendation

Probably instead of condition:

```
if msg.Amount > hostZone.StakedBal {
        return nil, sdkerrors.Wrapf(types.ErrInvalidAmount, "cannot unstake an amount g.t.
        ↪  staked balance on host zone: %d", msg.Amount)
```

```
    }
```

there should be:

```
if hostZoneUnbonding.NativeTokenAmount > hostZone.StakedBal {
        return nil, sdkerrors.Wrapf(types.ErrInvalidAmount, "cannot unstake an amount g.t.
        ↪  staked balance on host zone: %d", msg.Amount)
    }
```

where `hostZoneUnbonding.NativeTokenAmount` is assumed to be already updated:

```
hostZoneUnbonding.NativeTokenAmount += nativeAmount.Uint64()
```

In general, the unbonding and rebalancing logic is cumbersome, with many corner cases. It is thus not amendable to manual analysis due to many factors, which include, among others:

- frequency of `RedeemStake` calls
- the potential impact of the redemption rate
- hostZoneUnbonding statuses
- validators' staked amount

We recommend the unbonding and rebalancing logic to be formally modeled, verified, and model-based tested to ensure complete coverage of corner cases.

One chain redemption out-of-bounds may halt all chains

ID	IF-STRIDE-STAKEIBC-BEGINBLOCKER
Severity	High
Impact	High
Exploitability	Medium
Type	Implementation
Status	Unresolved

Involved artifacts

- stakeibc/abci.go

Description

The `BeginBlocker()` of `stakeibc` iterates over all Stride host zones, computes their redemption rate, and panics if it is outside of safety bounds.

```
// BeginBlocker of stakeibc module
func BeginBlocker(ctx sdk.Context, k keeper.Keeper, bk types.BankKeeper, ak
↪  types.AccountKeeper) {
    defer telemetry.ModuleMeasureSince(types.ModuleName, time.Now(),
    ↪  telemetry.MetricKeyBeginBlocker)


    // Iterate over all host zones and verify redemption rate
    for _, hz := range k.GetAllHostZone(ctx) {
        rrSafe, err := k.IsRedemptionRateWithinSafetyBounds(ctx, hz)
        if !rrSafe {
            panic(fmt.Sprintf("[INVARIANT BROKEN!!!] %s's RR is %s. ERR: %v", hz.GetChainId(),
            ↪  hz.RedemptionRate.String(), err.Error()))
        }
    }
}
```

Redemption rate is updated every Stride epoch (6 hours currently); the computation is done in the function UpdateRedemptionRates(), and the crux of it is represented by the formula `redemptionRate = (undelegatedBalance + stakedBalance + moduleAcctBalance)/stSupply`. The current safety bounds are hard-coded in stakeibc/types/params.go to be between 0.9 and 1.5.

Violation of redemption rate safety bounds for any Stride host chain, will lead to a panic in `BeginBlocker`, and thus will halt Stride operation for all Stride host chains.

Problem Scenarios

While the above computation seems reasonable *currently*, for the *current* set of host zones (Cosmos Hub, Juno, Osmosis, Stargaze), in the near future Stride aims to expand to around 50 host zones, as outlined in the Stride docs. Computation of the redemption rate depends on many parameters, not all of which might be under Stride's control; also their update logic is complicated and depends on e.g. success or failure of delivering IBC packets. Thus, the conditions violating the above safety bounds may either arise themselves for a particular chain (e.g. due to

inflation), or be engineered specifically to attack Stride. Especially worrying is that the (financial) resources needed for manipulation of the above parameters for one small chain will be also relatively small, while the effect of violation of the safety bounds for one chain will be global and devastating: a complete halt of Stride for all chains.

We rate the impact of this finding as **High**, because it may lead to the chain halt. We rate the exploitability of this finding as **Medium**; while in the current set of host zones the probability of the exploit is low, Stride expansion to 50 host zones brings a substantial risk that the exploitation conditions will arise within the next year. Thus, the vulnerability receives the overall severity score of **High**, and we recommend to address before expanding Stride to dozens of host zones.

Recommendation

The present finding surfaces the following problem: - Stride needs to be **safe** wrt. whatever faults or attacks happen on connected chains; - but Stride also needs to be **live** wrt. the chains that behave decently.

The current design of `BeginBlocker` represents the solution where liveness is sacrificed completely in favor of safety. The implemented solution for safety, however, is also not ideal, as halting the chain represents the extreme action, the consequences of which is hard to repair. There is however a space of solutions that allows to achieve both safety and liveness; below one such solution is outlined.

Implement per Zone Kill Switch

There should be a mechanism, a Kill switch, that allows to halt/suspend all operations *on a particular host zone* in case of any faults or attacks. The simplest solution for a kill switch from inside of Stride is probably the folllowing: - introduce the boolean field `Active` into HostZone; - add the functions `ActivateHostZone()`, `DeactivateHostZone()`, and `HostZoneActive()`, which, respectively, set the flag for a particular host zone to be true, false, or return whether it's true; - prefix all operations for a particular host zone with the precondition `if HostZoneActive(...)`

Taken together, this represents the on-chain circuitry needed for the kill switch.

Press the kill switch in BeginBlocker

Instead of using `panic` in `BeginBlocker`, replace the above code fragment with something along this lines:

```
// BeginBlocker of stakeibc module
func BeginBlocker(ctx sdk.Context, k keeper.Keeper, bk types.BankKeeper, ak
↪  types.AccountKeeper) {
    defer telemetry.ModuleMeasureSince(types.ModuleName, time.Now(),
    ↪  telemetry.MetricKeyBeginBlocker)


    // Iterate over all host zones and verify redemption rate
    for _, hz := range k.GetAllHostZone(ctx) {
        if k.HostZoneActive(ctx, hz) {
            rrSafe, err := k.IsRedemptionRateWithinSafetyBounds(ctx, hz)
            if !rrSafe {
                k.DeactivateHostZone(ctx, hz)
                k.Logger(ctx).Error(fmt.Sprintf("Redemption rate ouf of bounds for host zone
↪  %s", hz.ConnectionId))
            }
        }
    }
}
```

This will activate kill switch for one misbehaving host zone, but other host zones should continue to operate normally.

MsgActivateHostZone / MsgDeactivateHostZone for the Kill Switch "button"

Implement the "button" for the kill switch, via two new message types, `MsgDeactivateHostZone` and `MsgActivateHostZone`, which can be executed only with administrative privileges (via multisig): - `MsgDeactivateHostZone` is equivalent to `DeactivateHostZone()`, but can be submitted as a transaction. Having such external "button" is important, as not all possible faults or attacks can be foreseen at the construction time. In case some misbehavior is detected, the "button" will allow Stride operators to suspend operations on one chain, to investigate, and to have time for finding countermeasures. Such kill switch button can be also the blockchain interface for off-chain early detection / mitigation mechanisms. - `MsgActivateHostZone` is a "counter-button", which will allow to restart zone operations when corrective actions are taken.

The above mechanisms should enable per-chain safety and per-chain liveness, as well as allow to limit possible losses in case of faults or attacks.

IBC/tokens could be lost during IBC transfer to Delegation ICA

ID	IF-STRIDE-RECORDS-IBC_TOKENS_LOST_ON_STRIDE
Severity	High
Impact	Medium
Exploitability	High
Type	Implementation
Status	Unresolved

Involved artifacts

- x/records/module_ibc.go

Description

In the x/records module there is a custom implementation of the IBCModule interface `OnAcknowledgementPacket` function:

```
func (im IBCModule) OnAcknowledgementPacket(
    ctx sdk.Context,
    packet channeltypes.Packet,
    acknowledgement []byte,
    relayer sdk.AccAddress,
) error {
    im.keeper.Logger(ctx).Info(fmt.Sprintf("[IBC-TRANSFER] OnAcknowledgementPacket  %v",
↪   packet))
    // doCustomLogic(packet, ack)
    // ICS-20 ack
    var ack channeltypes.Acknowledgement
    if err := ibctransfertypes.ModuleCdc.UnmarshalJSON(acknowledgement, &ack); err != nil {
        im.keeper.Logger(ctx).Error(fmt.Sprintf("Error unmarshalling ack  %v", err.Error()))
        return sdkerrors.Wrapf(sdkerrors.ErrUnknownRequest, "cannot unmarshal ICS-20 transfer
        ↪  packet acknowledgement: %v", err)
    }

    // log the ack type
    switch resp := ack.Response.(type) {
    case *channeltypes.Acknowledgement_Result:
        im.keeper.Logger(ctx).Info(fmt.Sprintf("\t [IBC-TRANSFER] Acknowledgement_Result {%s}",
↪   string(resp.Result)))
    case *channeltypes.Acknowledgement_Error:
        im.keeper.Logger(ctx).Error(fmt.Sprintf("\t [IBC-TRANSFER] Acknowledgement_Error {%s}",
↪   resp.Error))
    default:
        im.keeper.Logger(ctx).Error(fmt.Sprintf("\t [IBC-TRANSFER] Unrecognized ack for packet
↪   {%v}", packet))
    }

    // Custom ack logic only applies to ibc transfers initiated from the `stakeibc` module
    ↪  account
```

```
    // NOTE: if the `stakeibc` module account IBC transfers tokens for some other reason in
    ↪  the future,
    // this will need to be updated
    err := im.keeper.ICACallbacksKeeper.CallRegisteredICACallback(ctx, packet, &ack)
    if err != nil {
        errMsg := fmt.Sprintf("Unable to call registered callback from records
↪   OnAcknowledgePacket | Sequence %d, from %s %s, to %s %s | Error %s",
            packet.Sequence, packet.SourceChannel, packet.SourcePort,
↪   packet.DestinationChannel, packet.DestinationPort, err.Error())
        im.keeper.Logger(ctx).Error(errMsg)
        return sdkerrors.Wrapf(icacallbacktypes.ErrCallbackFailed, errMsg)
    }

    return im.app.OnAcknowledgementPacket(ctx, packet, acknowledgement, relayer)
}
```

On the line 57 there is an explicit call to transfer module's `OnAcknowledgementPacket` function. As a sidenote, the outer record module's `OnAcknowledgementPacket` function will be called only when dealing with `TransferCallback` acks. In other words, this custom ack logic only applies to IBC transfers initiated from the `stakeibc` module account.

The problematic part could be prematurely return from record module's `OnAcknowledgementPacket` function due to potential ErrUnknownRequest and ErrCallbackFailed errors.

There is a similar issue in Stride's custom `onTimeoutPacket` function implementation:

```
func (im IBCModule) OnTimeoutPacket(
    ctx sdk.Context,
    packet channeltypes.Packet,
    relayer sdk.AccAddress,
) error {
    // doCustomLogic(packet)
    im.keeper.Logger(ctx).Error(fmt.Sprintf("[IBC-TRANSFER] OnTimeoutPacket  %v", packet))
    err := im.keeper.ICACallbacksKeeper.CallRegisteredICACallback(ctx, packet, nil)
    if err != nil {
        return err
    }
    return im.app.OnTimeoutPacket(ctx, packet, relayer)
}
```

Problem Scenarios

During IBC `SendTransfer` from Stride account to Delegation ICA, ibc/tokens are burned on Stride side. The intention with explicit call to `transfer.OnAcknowledgementPacket` and `transfer.OnTimeoutPacket` is to check ack from host zone and do appropriate custom logic - refunding burned ibc/tokens on Stride:

```
// OnAcknowledgementPacket responds to the the success or failure of a packet
// acknowledgement written on the receiving chain. If the acknowledgement
// was a success then nothing occurs. If the acknowledgement failed, then
// the sender is refunded their tokens using the refundPacketToken function.
func (k Keeper) OnAcknowledgementPacket(ctx sdk.Context, packet channeltypes.Packet, data
↪   types.FungibleTokenPacketData, ack channeltypes.Acknowledgement) error {
    switch ack.Response.(type) {
    case *channeltypes.Acknowledgement_Error:
        return k.refundPacketToken(ctx, packet, data)
    default:
        // the acknowledgement succeeded on the receiving chain so nothing
```

```
        // needs to be executed and no error needs to be returned
        return nil
    }
}

// OnTimeoutPacket refunds the sender since the original packet sent was
// never received and has been timed out.
func (k Keeper) OnTimeoutPacket(ctx sdk.Context, packet channeltypes.Packet, data
↪  types.FungibleTokenPacketData) error {
    return k.refundPacketToken(ctx, packet, data)
}
```

However, ibc/tokens will remain lost on the Stride chain, due to exiting these two custom functions (ErrUnknownRequest and ErrCallbackFailed) prior to sending the acknowledgment to the underlying core IBC module's functions.This will happen in cases of not being possible to - unmarshal the acknowledgment - which could be the appropriate reason for not refunding the tokens - perform ICA callbacks from the records module - which could lead to missing some important data in records store

The second situation could be defined by design as expected behavior, if Stride wishes to proceed working with failed ICACallbacks.

Recommendation

Consider justified reasons for not refunding the tokens and document them in the specification. The solution could be to move the underlying IBC Module's `OnAcknowledgementPacket` and `OnTimeoutPacket` on the beginning of the two custom implementations of the interface.

Users may not be able to redeem stake

ID	IF-STRIDE-STAKEIBC-REDEEM_STAKE
Severity	High
Impact	Medium
Exploitability	High
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/msg_server_redeem_stake.go

Description

Users can always redeem from Stride. When they select "redeeem" on the Stride website, Stride will initiate unbonding on the host zone. Once the unbonding period elapses, the users will receive native tokens in their wallets.

However, users cannot unstake an amount greater than staked balance on the host zone.

```
if msg.Amount > hostZone.StakedBal {
        return nil, sdkerrors.Wrapf(types.ErrInvalidAmount, "cannot unstake an amount g.t.
        ↪  staked balance on host zone: %d", msg.Amount)
    }
```

The problem with the code above is in comparing two different units; `msg.Amount` represents the input **stTokens** amount, while on the other hand `hostZone.StakedBal` describes host zone's total amount of **native** tokens.

Problem Scenarios

An `ErrInvalidAmount` error could be returned even if the user's redeeming amount doesn't exceed total host zone's staked amount. That's because stTokens are not 1-1 alligned with native tokens. There is `redemptionRate` defined as `nativeToken amount / stToken amount` which can fluctuate between 0.9 and 1.5 for current stride implementation, hard-coded in stakeibc/types/params.go.

Recommendation

Before comparing with `hostZone.StakedBal`, `msg.Amount` should be exchanged to the native tokens:

```
nativeAmount := sdk.NewDec(msg.Amount).Mul(hostZone.RedemptionRate).RoundInt()
```

After that, validation check for amount would be correct:

```
if nativeAmount > hostZone.StakedBal {
        return nil, sdkerrors.Wrapf(types.ErrInvalidAmount, "cannot unstake an amount g.t.
        ↪  staked balance on host zone: %d", msg.Amount)
    }
```

StakeExistingDepositsOnHostZones could be a bottleneck in case of many host zones

ID	IF-STRIDE-STAKEIBC-STAKE_EXISTING_DEPOSITS
Severity	Medium
Impact	Low
Exploitability	High
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/deposit_records.go

Description

Staking of existing deposits is not done by host zones but per epoch which contains all deposit records filtered by `DELEGATION_QUEUE` status. There is limitation of the number of staking deposit records to process per epoch:

```
maxDepositRecordsToStake := utils.Min(len(stakeDepositRecords), cast.ToInt(k.GetParam(ctx,
↪  types.KeyMaxStakeICACallsPerEpoch)))

for _, depositRecord := range stakeDepositRecords[:maxDepositRecordsToStake] {
...
}
```

where `KeyMaxStakeICACallsPerEpoch` is set to `100` by default.

Problem Scenarios

Probably there could be situations when the total number of deposits records per epoch may be greater than 100 (in the case of 40+ host zones e.g.). Then, some of them may not be processed and in the case of timeouts or failed acks, the processed ones will also get back to the delegation queue. This may cause some of deposit records being stuck and accumulated through previous epochs.

Recommendation

Consider doing the staking deposit records by host zone to make it easier to track the process and stats. Also think about having different limitations of staking ICA calls per each chain.

Failure to send IBC packets may lead to user funds freeze

ID	IF-STRIDE-STAKEIBC-UNBONDINGRECORDS
Severity	Medium
Impact	High
Exploitability	Low
Type	Implementation
Status	Unresolved

Involved artifacts

- stakeibc/keeper/unbonding_records.go

Description

In the function SweepAllUnbondedTokensForHostZone(), which is called at the beginning of every Stride "day" epoch from BeforeEpochStart() and SweepAllUnbondedTokens(); we find the following code fragment:

```
// Send the transaction through SubmitTx
_, err = k.SubmitTxsDayEpoch(ctx, hostZone.ConnectionId, msgs, *delegationAccount, REDEMPTION,
↪  marshalledCallbackArgs)
if err != nil {
    k.Logger(ctx).Info(fmt.Sprintf("Failed to SubmitTxs, transfer to redemption account on %s",
↪  hostZone.ChainId))
}
err = k.RecordsKeeper.SetHostZoneUnbondings(ctx, hostZone, epochUnbondingRecordIds,
↪  recordstypes.HostZoneUnbonding_EXIT_TRANSFER_IN_PROGRESS)
if err != nil {
    k.Logger(ctx).Error(err.Error())
    return false, 0
}
k.Logger(ctx).Info(fmt.Sprintf("Successfully completed unbonded token sweep ICA call for %s,
↪  %s, %v", hostZone.ConnectionId, hostZone.ChainId, msgs))
```

As can be seen, if an error is received from `SubmitTxsDayEpoch()`, the error is logged, but the execution continues; in particular, it updates the state of the relevant epoch unbonding records to `HostZoneUnbonding_EXIT_TRANSFER_IN_PROGRESS`. `SubmitTxsDayEpoch()` goes via ICAControllerKeeper.SendTx(), and eventually calls IBC send. There are many places in the long chain of calls that may fails due to variant reasons, and thus `SubmitTxsDayEpoch()` will fail as well.

Problem Scenarios

The above error will result in the following consequences: - Interchain Accounts IBC transaction won't be sent; - The unbonded funds won't be transferred to the Redemption ICA account; - RedemptionCallback() won't be called; and thus the status of unbonding records won't be set to `HostZoneUnbonding_CLAIMABLE`; - The funds that should have been transferred from Delegation ICA account to Redemption ICA account will be stuck at Delegation ICA account, and users won't be able to claim them.

We rate the impact of this finding as High, because it may lead to funds of many users locked, and users won't be able to claim them. We rate the exploitability of this finding as Low, as the possibility of ICA/IBC transfer failing

is relatively low. Due to the combination of the above two scores, vulnerability receives the overall severity score of Medium.

Recommendation

In the above code fragment, interrupt execution upon receiving an error from the `SubmitTxsDayEpoch()` function.

Consider isolating host zone operations

ID	IF-STRIDE-STAKEIBC-ZONEISOLATION
Severity	Informational
Impact	High
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- stakeibc/keeper/hooks.go
- stakeibc/abci.go

Description

In the current Stride StakeIBC architecture, in particular in the functions BeforeEpochStart() and BeginBlocker(), many actions are done via iteration over all host zones; we may call this architecture *horizontal*: all operations of a particular kind are tightly coupled together for all host zones.

As Stride plans to expand to ~50 host zones in the near future, we would like to propose the following *isolation property*, as desirable for the Stride implementation:

> **Isolation property**: whatever faults happen on one of the Stride host zones, should not influence Stride operations on other host zones.

The isolation property can be implemented via refactoring the Stride architecture from horizontal to *vertical*: here, the set of all operations for each host zone is done independently from the other host zones, and in particular they should be done in different blocks; thus e.g. the epochs for each host zone should be shifted wrt. each other. The difference between horizontal vs. vertical architecture is illustrated graphically in the image below.

Problem Scenarios

One of the main problems the current architecture poses is the existence of implicit assumptions that should be satisfied by the large volume of Stride code. E.g.: - the code should not panic (because it's executed in BeginBlocker/EndBlocker); - the code called from inside host zone iteration loops should not return prematurely, or break from those loops.

While the current codebase seem to adhere to those assumptions, as it is written by a few code developers, the assumptions may become violated unintentionally as the team grows. And if any of the assumptions is ever violated, the whole Stride operation for all host zones may break.

We rate the impact of this finding as High, because an unhandled error, or premature exit for any operation for one of the host zones will make the whole set of operations for all host zones to fail. We rate the exploitability of this finding as None, because in the current implementation we were not able to identify the way to trigger the problem. We would like to stress though that the exhaustive search and identification of all potential ways to exploit the problem is not feasible; e.g., an error may occur in other parts of the stack of dependencies: encoding/decoding, IBC, ICA, etc., which are outside of the scope of this report. Thus, we assign this finding the overall severity score of Informational: while not crucial now, we recommend to address it within the next year.

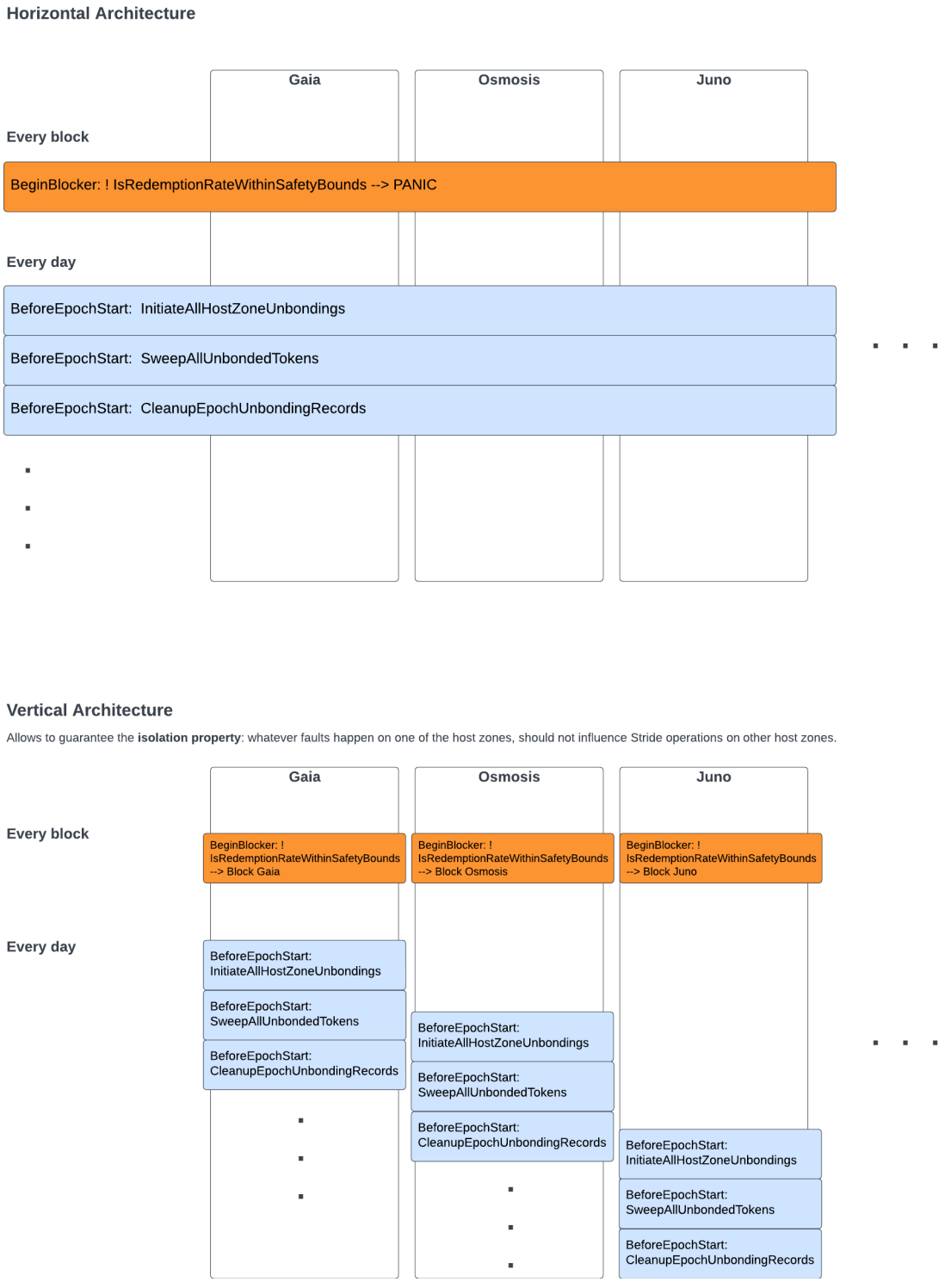

Figure 9: Stride: Horizontal vs Vertical Architecture

Recommendation

Consider refactoring the Stride StakeIBC implementation from *horizontal* to *vertical* architecture wrt. the handing of host zones. Besides that, consider documenting explicitly whatever important assumptions the code should satisfy (such as absence of panics).

GetHostZoneUnbondingMsgs need to be refactored

ID	IF-STRIDE-STAKEIBC-HOST_ZONE_UNBOINDING
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	PR created

Involved artifacts

- x/stakeibc/keeper/unbonding_records.go

Description

The job of `GetHostZoneUnbondingMsgs` function is to populate and return all `MsgUndelegate` messages as outlined below:

```
for _, valAddr := range utils.StringToIntMapKeys(valAddrToUnbondAmt) {
        valUnbondAmt := valAddrToUnbondAmt[valAddr]
        stakeAmt := sdk.NewInt64Coin(hostZone.HostDenom, valUnbondAmt)

        msgs = append(msgs, &stakingtypes.MsgUndelegate{
            DelegatorAddress: delegationAccount.GetAddress(),
            ValidatorAddress: valAddr,
            Amount:           stakeAmt,
        })

        splitDelegations = append(splitDelegations, &types.SplitDelegation{
            Validator: valAddr,
            Amount:    stakeAmt.Amount.Uint64(),
        })
    }
    ...
    return msgs, totalAmtToUnbond, marshalledCallbackArgs, epochUnbondingRecordIds, nil
```

`DelegatorAddress` is the address of Delegation ICA on host zone, and `ValidatorAddress` is the address of particular validator from which the token `Amount` will be undelegated.

The problematic part is the complexity that follows creating those messages.

`GetHostZoneUnbondingMsgs`:

- returns 5 different results (including error). On many places there are nasty returns like:
  - `go return nil, 0, nil, nil, sdkerrors.Wrap(types.ErrIntCast, errMsg) return nil, 0, nil, nil, sdkerrors.Wrap(types.ErrNoValidatorAmts, errMsg) return nil, 0, nil, nil, nil ...`
- has 4 for loops (including 2x iterations over the validator set)
- doesn't have single responsibility; instead it's doing many things:
  - Calculate total unbonding amount
  - Populate mapping (validator address -> unbond amount)
  - Dealing with overflows

  - Creating MsgUndelegate for each (delegator, validator) pair
  - Creating undelegate callback

Problem Scenarios

Code maintenance might become difficult.

Recommendation

- Pull the code that calculates total amount to unbond out from the function. The function is only interested in the final result: `totalAmtToUnbond`.

```
 for _, epochUnbonding := range k.RecordsKeeper.GetAllEpochUnbondingRecord(ctx) {
       hostZoneRecord, found := k.RecordsKeeper.GetHostZoneUnbondingByChainId(ctx,
↪  epochUnbonding.EpochNumber, hostZone.ChainId)
       if !found {
           errMsg := fmt.Sprintf("Host zone unbonding record not found for hostZoneId %s in
↪  epoch %d",
               hostZone.ChainId, epochUnbonding.GetEpochNumber())
           k.Logger(ctx).Error(errMsg)
           continue
       }
       // mark the epoch unbonding record for processing if it's bonded and the host zone
       ↪  unbonding has an amount g.t. zero
       if hostZoneRecord.Status == recordstypes.HostZoneUnbonding_UNBONDING_QUEUE &&
       ↪  hostZoneRecord.NativeTokenAmount > 0 {
           totalAmtToUnbond += hostZoneRecord.NativeTokenAmount
           epochUnbondingRecordIds = append(epochUnbondingRecordIds,
↪  epochUnbonding.EpochNumber)
           k.Logger(ctx).Info(fmt.Sprintf("[SendHostZoneUnbondings] Sending unbondings, host
↪  zone: %s, epochUnbonding: %v", hostZone.ChainId, epochUnbonding))

       }
   }
```

- Calculating and handling the overflow should also be moved from the function:

```
 if overflowAmt > 0 { // if we need to reallocate any weights
       for _, validator := range validators {
           valAddr := validator.GetAddress()
           ...
       }
   }
   if overflowAmt > 0 {
       errMsg := fmt.Sprintf("Could not unbond %d on Host Zone %s, unable to balance the
↪  unbond amount across validators",
           totalAmtToUnbond, hostZone.ChainId)
       ...
   }
```

It's not responsibility of the `GetHostZoneUnbondingMsgs` to deal with unbonding amount overflows. - Rename `stakeAmt` to `unstakeAmt`:

```
 for _, valAddr := range utils.StringToIntMapKeys(valAddrToUnbondAmt) {
       valUnbondAmt := valAddrToUnbondAmt[valAddr]
       stakeAmt := sdk.NewInt64Coin(hostZone.HostDenom, valUnbondAmt)
```

```
        msgs = append(msgs, &stakingtypes.MsgUndelegate{
            DelegatorAddress: delegationAccount.GetAddress(),
            ValidatorAddress: valAddr,
            Amount:           stakeAmt,
        })

        splitDelegations = append(splitDelegations, &types.SplitDelegation{
            Validator: valAddr,
            Amount:    stakeAmt.Amount.Uint64(),
        })
    }
```

Variable err not assigned but used inside if condition

ID	IF-STRIDE-STAKEIBC-HOST_ZONE_UNBOINDING_2
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/unbonding_records.go

Description

Variable `err` doesn't have associated value but it is used inside if condition:

```
for _, validator := range validators {
        valAddr := validator.GetAddress()
        valUnbondAmt := newUnbondingToValidator[valAddr]
        currentAmtStaked := validator.GetDelegationAmt()
        if err != nil {
            errMsg := fmt.Sprintf("Error fetching validator staked amount %d: %s",
↪  currentAmtStaked, err.Error())
            k.Logger(ctx).Error(errMsg)
            return nil, 0, nil, nil, sdkerrors.Wrap(types.ErrNoValidatorAmts, errMsg)
        }
        ...
    }
```

Function GetDelegationAmt() doesn't return any errors.

Problem Scenarios

Existing of dead code inside for loop. Even though `err` should never be different than `nil` in the current implementation, reading the code could lead to wrong conclusions.

Recommendation

Remove `if` condition from the code in Description. Handle the 0 delegation amount (`currentAmtStaked`) case if there is a need for that.

Error handling should be reviewed

ID	IF-STRIDE-STAKEIBC-ERRORHANDLING
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/hooks.go
- x/icacallbacks/keeper/keeper.go

Description

`InitiateAllHostZoneUnbondings` and `SweepAllUnbondedTokens` functions have return values defined but these are not handled in the `if` block:

```
if epochIdentifier == "day" {
    // here, we process everything we need to for redemptions
    k.Logger(ctx).Info(fmt.Sprintf("Day %d Beginning", epochNumber))
    // first we initiate unbondings from any hostZone where it's appropriate
    k.Logger(ctx).Info("InitiateAllHostZoneUnbondings")
    k.InitiateAllHostZoneUnbondings(ctx, epochNumber)
    // then we check previous epochs to see if unbondings finished, and sweep the tokens if so
    k.Logger(ctx).Info("SweepAllUnbondedTokens")
    k.SweepAllUnbondedTokens(ctx)
    ...
}
```

On the other hand, `CallRegisteredICACallback` returns nil in both cases, the successful one (callback is processed and callback data is removed), and when there is no actual callback data. It's clear that some IBC or ICA transactions don't need to have callback defined but the caller of the function couldn't know if the callback was called and processed or the callback data hasn't been defined.

```
func (k Keeper) CallRegisteredICACallback(ctx sdk.Context, modulePacket channeltypes.Packet,
↪   ack *channeltypes.Acknowledgement) error {
    callbackDataKey := types.PacketID(modulePacket.GetSourcePort(),
↪   modulePacket.GetSourceChannel(), modulePacket.Sequence)
    callbackData, found := k.GetCallbackDataFromPacket(ctx, modulePacket, callbackDataKey)
    if !found {
        return nil
    }
    callbackHandler, err := k.GetICACallbackHandlerFromPacket(ctx, modulePacket)
    if err != nil {
        k.Logger(ctx).Error(fmt.Sprintf("GetICACallbackHandlerFromPacket %s", err.Error()))
        return err
    }

    // call the callback
```

```
    if (*callbackHandler).HasICACallback(callbackData.CallbackId) {
        k.Logger(ctx).Info(fmt.Sprintf("Calling callback for %s", callbackData.CallbackId))
        // if acknowledgement is empty, then it is a timeout
        err := (*callbackHandler).CallICACallback(ctx, callbackData.CallbackId, modulePacket,
↪   ack, callbackData.CallbackArgs)
        if err != nil {
            errMsg := fmt.Sprintf("Error occured while calling ICACallback (%s) | err: %s",
↪   callbackData.CallbackId, err.Error())
            k.Logger(ctx).Error(errMsg)
            return sdkerrors.Wrapf(types.ErrCallbackFailed, errMsg)
        }
    } else {
        k.Logger(ctx).Error(fmt.Sprintf("Callback %v has no associated callback",
↪   callbackData))
    }

    // remove the callback data
    k.RemoveCallbackData(ctx, callbackDataKey)
    return nil
}
```

Problem Scenarios

Caller functions could have misleading information based on return value(s) of the calling functions.

Recommendation

For `InitiateAllHostZoneUnbondings` think about handling the case when there are some failed unbondings, or similar in `SweepAllUnbondedTokens` if there are failed sweeps from Delegation ICA to Redemption ICA.

In the `CallRegisteredICACallback` at least consider adding an info log before returning `nil` if it's expected that some acks do not have an associated callback. The other aproach is to add dummy callback for every IBC tx.

Different conditions for max number of validators to rebalance

ID	IF-STRIDE-STAKEIBC-REBALANCEVALIDATORS
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/msg_server_rebalance_validators.go
- x/stakeibc/types/message_rebalance_validators.go

Description

There are two different places in the code where conditions for maximum number of validators to rebalance are checked and they don't really match. First one is in the function ValidateBasic():

```
if (msg.NumRebalance < 1) || (msg.NumRebalance > 10) {
        return sdkerrors.Wrapf(sdkerrors.ErrInvalidRequest, fmt.Sprintf("invalid number of
        ↪  validators to rebalance (%d)", msg.NumRebalance))
    }
```

and the other one is part of RebalanceValidators() implementation.

```
if maxNumRebalance < 1 {
        k.Logger(ctx).Error(fmt.Sprintf("Invalid number of validators to rebalance %d",
↪  maxNumRebalance))
        return nil, types.ErrInvalidNumValidator
    }
    if maxNumRebalance > 4 {
        k.Logger(ctx).Error(fmt.Sprintf("Invalid number of validators to rebalance %d",
↪  maxNumRebalance))
        return nil, types.ErrInvalidNumValidator
    }
```

"Magic numbers" are also used inside if conditions.

Problem Scenarios

`MsgRebalanceValidators` will be processed successfully by `ValidateBasic()` for any NumRebalance value in {5..10}, but it will fail later on msg execution.

Recommendation

Choose which values could be taken for number of validators to rebalance and do the check only at one place (e.g. in `ValidateBasic()`).

Replace numbers with named constants (e.g. `MIN_NUM_REBALANCE` and `MAX_NUM_REBALANCE`)

Combine two conditions into single expression:

```
if (msg.NumRebalance < MIN_NUM_REBALANCE) || (msg.NumRebalance > MAX_NUM_REBALANCE) {
        return sdkerrors.Wrapf(sdkerrors.ErrInvalidRequest, fmt.Sprintf("invalid number of
        ↪  validators to rebalance (%d)", msg.NumRebalance))
    }
```

Redemption rate limits are hardcoded

ID	IF-STRIDE-STAKEIBC-REDEMPTIONRATELIMITS
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/types/params.go

Description

The way redemption rate works is it uses formula:

```
redemptionRate = (undelegatedBalance + stakedBalance + moduleAcctBalance)/stSupply
```

where the amount inside parentheses represents all the native tokens that Stride protocol owns (ATOMs for Cosmos Hub), and `stSupply` is staked tokens amount (stATOMs for Cosmos Hub). In the downside, slashing should be limited to ~5% so redemption rate could go down for 5% (normalized to 0.95). On the upside it's limited by the inflation (for Cosmos Hub it's 20% per year), so the expectation is for redemption rate to go up to 1.2 after one year.

Right now, the limit range is hardcoded in stakeibc/types/params.go to be between 0.9 and 1.5.

Problem Scenarios

Different chains have different inflation rates and slashing mechanisms. Also, the redemption rate limits are by nature time dependent.

Recommendation

Each host zone should dynamically specify redemption rate limits. So, consider shifting them through time.

Documentation is not updated (including Linux support)

ID	IF-STRIDE-DOCUMENTATION
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- https://docs.stride.zone/docs

Description

The documentation contains pretty well technical architecture diagrams which explain three main data flow processes:

- deposit and liquid stake
- epoch delegation and reinvestment
- unbonding

But some parts of the epoch delegation and reinvestment diagram need to be updated:

- Reward Account probably should be removed and arrow no.3 may point to Withdraw ICA directly.

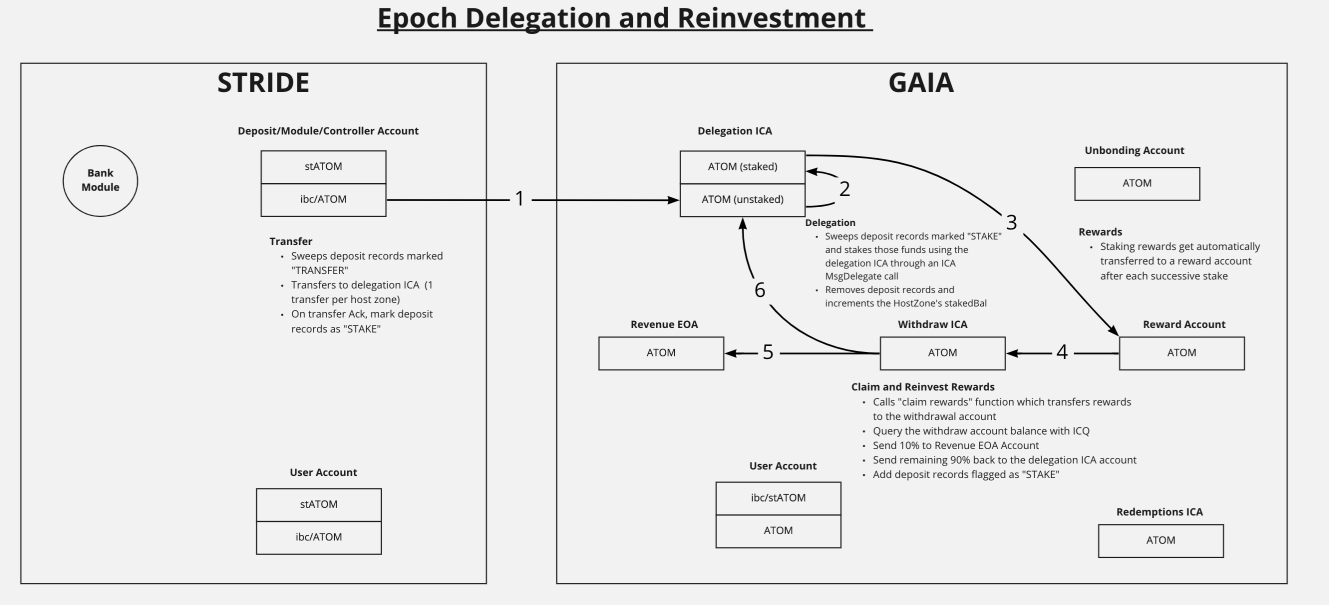

Figure 10: Alt text

- Function SetWithdrawalAddressOnHost changes default withdraw address (Delegation ICA address) to Withdraw ICA using `MsgSetWithdrawAddress` from `x/distribution` module:

```
msgs = append(msgs, &distributiontypes.MsgSetWithdrawAddress{DelegatorAddress:
↪  delegationIca.GetAddress(), WithdrawAddress: withdrawalIcaAddr})
```

so the "Reward Account" is an internal account in the `dist` module and is not important for the big picture.

- There are deprecated enums: `TRANSFER` and `STAKE`

Up to this point, dockernet has only been tested on MacOS. In other words, nothing hasn't been done or tested on Linux. The audit team experienced a lot of installing problems using Linux machines so the solution rapidly appeared; the stride team provided a dedicated branch for Linux users: linux-dockernet and PR which introduces changes to make dockernet compatible with Linux.

Problem Scenarios

Out-of-date technical diagrams could lead to wrong conclusions about the data flow. Additionally, Linux users may feel some inconvenience while installing the chain.

Recommendation

- Rename `Revenue EOA` to either `Revenue ICA` or `Fee ICA` to be consistent with other ICA accounts on the host zone
- Remove the first sentence under the "Claim and Reinvest Rewards" section. There is no such thing as a "claim rewards" function anymore.
- On the epoch delegation diagram make use of newly added enums for deposit records:
  - `TRANSFER_QUEUE`
  - `TRANSFER_IN_PROGRESS`
  - `DELEGATION_QUEUE`
  - `DELEGATION_IN_PROGRESS`

to clearly distinguish the current state of each deposit record instead of the former ones (`TRANSFER` and `STAKE`). - Similar thing for unbonding flow; mention the HostZoneUnbonding possible status: - `UNBONDING_QUEUE` - `UNBONDING_IN_PROGRESS` - `EXIT_TRANSFER_QUEUE` - `EXIT_TRANSFER_IN_PROGRESS` - `CLAIMABLE`

- Replace the non-used `ignite chain serve` command which belongs to the Installing Stride section from docs with the appropriate one.
- Dockernet linux support : already done.

Coding style recommendations

ID	IF-STRIDE-CODINGSTYLE
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- x/stakeibc/keeper/msg_server_redeem_stake.go line 129-135
- x/stakeibc/keeper/host_zone.go
- x/stakeibc/keeper/hooks.go line58
- x/stakeibc/keeper/hooks.go line 167
- x/stakeibc/keeper/msg_server_redeem_stake.go line 23-26
- x/stakeibc/keeper/icacallbacks_reinvest.go

Description

- Redundant casting:

```
    stTokenAmount, err := cast.ToUint64E(msg.Amount)
    if err != nil {
        errMsg := fmt.Sprintf("Could not convert redemption amount to int64 in redeem stake |
↪   %s", err.Error())
        k.Logger(ctx).Error(errMsg)
        return nil, sdkerrors.Wrapf(types.ErrIntCast, errMsg)
    }
    hostZoneUnbonding.StTokenAmount += stTokenAmount
```

Both `msg.Amount` and `hostZoneUnbonding.StTokenAmount` are already `uint64`.

- Redundant `else` keyword:

```
    if amt >= 0 {
        amt, err := cast.ToUint64E(amt)
        if err != nil {
            k.Logger(ctx).Error(fmt.Sprintf("Error converting %d to uint64", amt))
            return false
        }
        val.DelegationAmt = val.GetDelegationAmt() + amt
        return true
    } else {
        absAmt, err := cast.ToUint64E(-amt)
        if err != nil {
            k.Logger(ctx).Error(fmt.Sprintf("Error converting %d to uint64", amt))
            return false
        }
        if absAmt > val.GetDelegationAmt() {
            k.Logger(ctx).Error(fmt.Sprintf("Delegation amount %d is greater than validator %s
↪   delegation amount %d", absAmt, valAddr, val.GetDelegationAmt()))
```

```
            return false
        }
        val.DelegationAmt = val.GetDelegationAmt() - absAmt
        return true
    }
```

- Using hardcoded string:

```
    if epochIdentifier == "day" {
```

- Code duplication:

```
sender, err := sdk.AccAddressFromBech32(msg.Creator)
    if err != nil {
        return nil, sdkerrors.Wrapf(sdkerrors.ErrInvalidAddress, "creator address is invalid:
        ↪  %s. err: %s", msg.Creator, err.Error())
    }
```

The code above repeats in ValidatorBasic() as well as in msgServer's implementation of `RedeemStake`.

- Naming optimization:

```
    amount := reinvestCallback.ReinvestAmount.Amount
    denom := reinvestCallback.ReinvestAmount.Denom
```

- MintStAsset does two things: minting and sending tokens. # Problem Scenarios

Recommendation

- Remove redundant type conversion statement.
- In the above, the else statement can be safely removed because its if clause returns from the method. Thus, even without the else, there's no way you'll be able to proceed past the if clause body.
- Avoid using hardcoded strings (`"day"`), make use of named constants instead (`DAY_EPOCH`).
- Remove error checking in msgServer's implementation of `RedeemStake`. It's been already checked in ValidatorBasic().
- Change the name of `ReinvestAmount` type to `ReinvestCoin`, because it contains `Amount` and `Denom` fields.
- According to SRP sending `stTokens` to user's Stride account shouldn't be the responsibility of MintStAsset function. Take the sending part out of the function and place it inside LiquidStake.

Stride lacks various test cases

ID	IF-STRIDE-TESTING-DEFICIENCY
Severity	Informational
Impact	None
Exploitability	None
Type	Test
Status	Unresolved

Involved artifacts

- scripts/tests/gaia_tests.bats
- epochs/README.md

Description

Stride has a decent unit test coverage, but lacks some further testing such as automated fuzz testing. For example, "liquid stake" and "redeem stake" methods are good candidates for fuzzing. Integration tests are written sequentially, so that each test depends on the previous tests and examines the chain state at a point in time. The written integration tests are passing which means main functionalities are working fine, but it looks like only "happy paths" are covered. scripts/tests/gaia_tests.bats

There are no tests which simulate the situations when something could potentially go wrong, such as system behaviour when timeouts happen, etc. A lot of code in Stride is running periodically using epochs and basically all the logic is dependent on epochs. There are no tests which checks the behaviour of the system when configuring various epoch durations even though there are warnings issued:

epochs/README.md

Recommendation

The recommendation is to add fuzz testing which will provide invalid, unexpected or random data as inputs. This could help exposing vulnerabilities and corner cases which weren't considered when writing unit tests. Consider testing stride functionalities with different epoch durations. Also, it looks like there is an assumption that relayers will always work very efficiently. It would be great if there is a possibility to check the behaviour of system with adding tests based on possible arbitrary behaviour of relayers. Finally, there should be some tests which are a result of scalability analysis, since there is a plan to add more Host Zones. Try to check the behaviour of the system with increasing number of Host Zones.

Various kinds of observations

ID	IF-STRIDE-OBSERVATIONS
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Unresolved

Description

- **There are no invariants defined in Stride codebase**. In the context of the Cosmos SDK, an Invariant is a function that checks for a particular invariant at the end of each block. These functions are useful to detect bugs early on and act upon them to limit their potential consequences (e.g. by halting the chain). They are also useful in the development process of the application to detect bugs via simulations.
- **Simulation tests not implemented**. The Cosmos SDK offers a full-fledged simulation framework to fuzz-test every message defined by a module. Although Stride modules contain scaffolded skeletons for simulation messages, they are not implemented (e.g. LiquidStake msg):

```
  func SimulateMsgLiquidStake(
ak types.AccountKeeper,
bk types.BankKeeper,
k keeper.Keeper,
  ) simtypes.Operation {
return func(r *rand.Rand, app *baseapp.BaseApp, ctx sdk.Context, accs []simtypes.Account,
↪  chainID string,
    ) (simtypes.OperationMsg, []simtypes.FutureOperation, error) {
  simAccount, _ := simtypes.RandomAcc(r, accs)
  msg := &types.MsgLiquidStake{
    Creator: simAccount.Address.String(),
  }

  // TODO: Handling the LiquidStake simulation

  return simtypes.NoOpMsg(types.ModuleName, msg.Type(), "LiquidStake simulation not
  ↪  implemented"), nil, nil
    }
  }
```

- `HostZoneUnbonding`'s status `EXIT_TRANSFER_QUEUE` was set twice, before and after burning `stTokens` during `UndelegateCallback`.
- `SetWithdrawalAddressOnHost` is being called on every epoch.
- Flags: `HostZoneUnbonding_UNBONDING_IN_PROGRESS` and `HostZoneUnbonding_EXIT_TRANSFER_IN_PROGRESS` seem to be unused.

Problem Scenarios

Recommendation

- Think about adding some invariants. Describe and check conditions on Stride which should be fulfilled at every block execution or on every epoch.
- If there is a need for fuzzy testing, you could utilize cosmos sdk simulator by implementing messages of interest. These are simulated with random field values. The sender of the operation is also assigned randomly. The simulator can test parameter changes at random as well. Finally, you could create a randomized genesis file.
- Consider representing `amount` fields with string type (if their type `int64` or `uint64`). The way protobuf json marshaling works, it encodes the `uint64` as a `string`. This also allows for larger amounts such as `uint256` to be sent across chains. An example of this approach can be found in proto/ibc/applications/transfer/v2/packet.proto:

```
message FungibleTokenPacketData {
  // the token denomination to be transferred
  string denom = 1;
  // the token amount to be transferred
  string amount = 2;
  // the sender address
  string sender = 3;
  // the recipient address on the destination chain
  string receiver = 4;
}
```

- Set `HostZoneUnbonding`'s status `EXIT_TRANSFER_QUEUE` either before or after burning `stTokens` during `UndelegateCallback`.
- `SetWithdrawalAddressOnHost` should not be called on every epoch.
- Define only necessary flags for `HostZoneUnbonding_Status`. Probably `HostZoneUnbonding_UNBONDING_IN_PROGRESS` and `HostZoneUnbonding_EXIT_TRANSFER_IN_PROGRESS` should be removed.