coinspect

You build, we defend.

babylon

Source Code Audit

V4

October 2025

Babylon v4
Source Code Audit

Version: v251029 Prepared for: Babylon October 2025

Security Assessment

1. Executive Summary

In **October 2025**, **Babylon Labs** engaged Coinspect to perform a Source Code Audit of the V4 version of the Babylon Genesis node and related projects such as its finality provider, vigilante and web application.

✓ Solved	⚠ Caution Advised	✗ Resolution Pending
High 4	High 0	High 0
Medium 2	Medium 0	Medium 0
Low 4	Low 0	Low 0
No Risk 8	No Risk 0	No Risk 0
Total **18**	Total **0**	Total **0**

2. Summary of Findings

This section provides a concise overview of all the findings in the report grouped by remediation status and sorted by estimated total risk.

2.1 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
V4-CORE-001	Attacker can steal all rewards after score ratio update	High
V4-CORE-006	Attacker can cause total BABY stake to be miscounted	High
V4-CORE-007	Attacker can steal all rewards by restaking BABY tokens	High
V4-WEB-004	Denial of Service due to unbounded mongoDb query	High
V4-CORE-005	Finality provider can double sign due to race condition	Medium
V4-WEB-001	Adversaries can abuse Chainalysis API usage quota due to missing rate limiting	Medium
V4-CORE-002	Integer overflow in derivation path parsing	Low
V4-CORE-004	Finality provider can generate invalid 0×00 private keys	Low
V4-WEB-003	Lack of rate limiting and ownership verification in terms acceptance endpoint	Low
V4-WEB-006	Malformed staking expansion transaction on API	Low

	failure	
V4-CORE-003	Finality provider does not check for error on writing public randomness	None
V4-CORE-008	Potential misaccounting in RemoveBondedSats due to underflow safeguard	None
V4-WEB-002	Potentially inaccurate unbonding statistics	None
V4-WEB-005	Usage of a fixed gas price might lead to stuck transactions	None
V4-WEB-007	Geo-block can be bypassed	None
V4-WEB-008	Redundant transaction confirmation logic	None
V4-WEB-009	Ineffective transaction hash validation in staking expansion	None
V4-WEB-010	Stale tipHeight can lead to transaction failure or invalid staking transaction	None

3. Scope

The scope encompassed several repositories dealing with the Babylon Genesis chain, its supporting off-chain software and also web applications.

Most of the scope was defined as pull requests on the Github repository of each project.

- https://github.com/babylonlabs-io/babylon
  - `x/costaking` at PR https://github.com/babylonlabs-io/babylon/pull/1755
  - Epoching module changes at PR https://github.com/babylonlabs-io/babylon/pull/1739
  - Power distribution event changes at PR https://github.com/babylonlabs-io/babylon/pull/1735
  - Simplification of selective slashing at PR https://github.com/babylonlabs-io/babylon/pull/1740
  - Update of Cosmos Stack, including
    - Bump to SDK 0.53 at PR #1736 and PR #1747.
  - Public randomness commit lookup optimization at PR #1745.
  - Disallowing fee grants on refundable transactions at PR #1746.
  - Enhancing queued message API at PR #1741.
  - Stake expansion feature at PR #1231, with further fix commits
    - PR #1748
    - PR #1749
    - PR #1750
    - PR #1751
- https://github.com/babylonlabs-io/vigilante
  - Stake expansion feature at PR #380 and PR #406
- https://github.com/babylonlabs-io/covenant-emulator/
  - Stake expansion feature at #127
  - Stake expansion fix at #128
  - Fix to avoid batch-failure when a single message fails at #112.

Some repositories in scope did not have specific pull requests defined, these were:

- https://github.com/babylonlabs-io/finality-provider
  - Refactor from `v1.1.0-rc.1` (commit `be550f23abe7ce154336717791b30cdaca0fc318`) to `v2.0.0-rc.5` (commit `840dff5d1c7214123fe3f21ad37f678e4b557616`).

- https://github.com/babylonlabs-io/btc-staking-ts at tag `v2.5.7` (commit `93b0a2f591d976dab22b3005aedfe3bd16540cd6`).
- https://github.com/babylonlabs-io/babylon-toolkit at tag `v1.2.66` (commit `8c1a639a83f7dbea8af2a6da084932adf8ee5fb0`).
  - `babylon-toolkie` only directory under review was `services/simple-staking`, and only the code path where `IsTimelockRenewalEnabled` is set to `true`.
- https://github.com/babylonlabs-io/staking-api-service at tag `v3.0.0-rc.1` (commit `7769bcc060915588fbd4855fc1160cfad01796df`).
- https://github.com/babylonlabs-io/babylon-staking-indexer at tag `v3.0.0-rc.1` (commit `12472fedf4211b1976bed40dfc302a533ac68089`)

It is worth pointing out that this engagement built on previous work done by Coinspect on an experimental version of Babylon with a different threat model than the one for `v4`. Issues that are relevant for the current scope are reproduced here, with the addition of new findings related to the specific features introduced in the PRs and commits specified in the scope above.

3.1 Fixes review

The fixes review was conducted on specific PRs shared by Babylon Labs team. Each issue has a specific status detailing its fix, if existing.

4. Assessment

Babylon Version 4 major addition is the introduction of `x/costaking`, a new module intended to provide improved rewards to stakers who secure Babylon both by staking native BABY tokens to CometBFT Validators and staking BTC to finality providers. A set of high-severity issues were identified by Coinspect in this component (see `V4-CORE-001`, `V4-CORE-007` and `V4-CORE-008`) and at least one other by Babylon Labs during this review; which indicates that this module required improved adversarial and edge case tests. Babylon Labs team has added tests along with the fixes for the vulnerabilities reported (see section `3.1 Fixes review`) which are intended to prevent reintroducing the reported vulnerabilities and improving the security posture of the project.

The other major addition to the Babylon ecosystem is the stake extension feature, which allows a BTC staker to renew the timelock of their stake or increase the staked BTC without the need to unstake and experience interruptions in their BTC staking reward accrual.

Besides, a new security-relevant spam-prevention mode was added to the `x/epoching` module. `x/epoching` is the module in charge of timing validator set changes in Babylon Genesis so as to prevent desyncs with the checkpoints sent to Bitcoin.

4.1 Security assumptions

- The Babylon governance and covenant quorum is honest
- Core Cosmos primitives work correctly
- The Bitcoin blockchain is safe and live
- Majority of Babylon Genesis validators are honest
- Finality provider set is decentralized

5. Detailed findings

V4-CORE-001

Attacker can steal all rewards after score ratio update

Status
Solved

✓

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`x/costaking/keeper/msg_server.go`

Description

An attacker can empty the `costaking` module's bank after a score ratio update by Babylon governance, because the score ratio update does not reset each costaker's `StartPeriodCumulativeReward`. This means costakers retroactively calculate rewards using their new score for old periods. The first costaker to realize this can then claim rewards up to the new limit, effectively stealing from other costakers.

To understand the issue, one should consider how `x/costaking` rewards work in Babylon. There are three structures that are relevant: `CostackerRewardsTracker`, `CurrentRewards` and `HistoricalRewards`. The system also has a score to balance BABY and BTC stakes.

`CostakerRewardsTracker` is a per-costaker struct that contains how many active satoshis and baby the staker has in the system, their total score and their `StartPeriodCumulativeReward`. The `StartPeriodCumulativeReward` is a critical piece of information: it is used by the system to calculate the total rewards owed to the staker. Conceptually, it indicates the last period in which the staker's state was modified. It is used to calculate rewards assuming no changes were made to the staking state since them.

`CurrentRewards` is a global tracker which collects to the total rewards in this period (in the `Rewards` field), the `Period` and the `TotalScore`, the sum of all costaker's scores.

`HistoricalRewards` are used to keep the rewards for each period. When a a global period is incremented, a new historical reward is created with the `CumulativeRewardsPerScore` of the recently incremented period plus the `CumulativeRewardsPerScore` of the previous period. As such `HistorialRewards` is a monotonically increasing tracker of the rewards accumulated up to a period.

When a costaker wishes to withdraw their rewards, they need to call `WithdrawRewards` of the `x/incentive` module. This calls `costakerModifiedScoreWithPreInitalization` which increments both the global period and calculates the amount owed via `CalculateCostakerRewards` by using the `HistoricalRewards`. As they are accumulative, the system can calculate `(HistorialRewards[LastPeriod] - HistoricalRewards[costaker.StartPeriodCumulativeReward) * costaker.TotalScore` to get how much is owed to the staker per score. It then sends the appropriate coins from the `costaking` bank to the `incentive` bank, and from there will be moved to the withdrawer's address. Note that `costakerModifiedScoreWithPreInitalization` is also called everytime the costaker is modified (by adding or removing stake) - which also updates their `LastPeriod`. This is needed so that a costaker's `Period` represents a frozen amount of stake and score over which the reward calculation can be performed.

As mentioned, the `HistoricalRewards` calculation uses the `TotalScore` of the costaker. This `TotalScore` is normally modified by the `costakerModified` method when their amount of BTC or BABY staked changes, and is a sort of virtual staked amount used to calculate the rewards: it is set to be the `min(active satoshi, active_baby / ScoreRatioBtcToBaby)`, where the `ScoreRatioBtcToBaby` is set by Babylon's governance. This score is needed as the prices and supplies of BTC and BABY are not equal, so the `ScoreRatioBtcToBaby` incentivizes costakers to provide a certain ratio of each coin to maximize costaking rewards. Put simply: a costaker is incentivized to get a near equal amount of `active_satoshi` and `active_baby / ScoreRatioBtcToBaby` to maximize their costaking rewards.

The issue is that the `TotalScore` can also be manually modified by Babylon's governance when updating the `ScoreRatioBtcToBaby` parameter. This breaks an invariant held by `CalculateCostakerRewards`. In particular, the inner function `calculateCoStakerRewardsBetween` assumes that `costaker.TotalScore` was a constant for all periods between `StartPeriodCumulativeReward` and the last period. But this is not the case: an update of the `ScoreRatioBtcToBaby` by the governance breaks this invariant, as it calls `UpdateScore` for all costakers without calling `costakerModifiedScoreWithPreInitalization`, which is in charge of making sure the state of the system is consistent when a parameter relevant to calculate rewards changes.

This means that after a score ratio, an staker can call `WithdrawRewards`, which will call `CalculateCostakerRewards`, which will use the *new staker's score* given by the *new ratio* to calculate the owed rewards. There are two scenarios here:

1. The new ratio benefits the staker, effectively inflating the amount of rewards owed to them
2. The new ratio reduces the score of the staker, reducing the amount of rewards owed to them

Scenario (1) allows an staker to withdraw more rewards than they are owed. What is more: because the `costaking` banking module does not actually have the funds to back this new rewards calculation, they can withdraw rewards that should belong to other costakers. In the worst case scenario, the attacker could drain the whole costaking banking module.

Scenario (2) gives undue power to the Babylon governance, which could retroactively reduce rewards for all costakers.

The issue is considered of high impact due to the possibility of scenario (1) which would allow an attacker to take other's rewards. The likelihood is high because, while it requires a change through the governance, it does not require the governance to be malicious. What is more, it is very likely the score ratio will need to be updated at some point due to price fluctuations.

Recommendation

Update the `StartPeriodCumulativeReward` for all stakers after a `ScoreRatioBtcToBaby` update.

Status

Fixed at PR #1780.

Proof of concept

Intended for a new file in the `x/costaking/keeper` module:

```go
package keeper

import (
	"testing"

	sdkmath "cosmossdk.io/math"
	"github.com/stretchr/testify/require"

	appparams "github.com/babylonlabs-io/babylon/v4/app/params"
	"github.com/babylonlabs-io/babylon/v4/testutil/datagen"
	"github.com/babylonlabs-io/babylon/v4/x/costaking/types"
	sdk "github.com/cosmos/cosmos-sdk/types"
)

func TestUpdateAllCostakersScore_VulnerabilityNoMocks(t *testing.T) {
	k, ctx := NewKeeperWithMockIncentiveKeeper(t, nil)

	// Setup two costakers
	attacker := datagen.GenRandomAddress()
	victim := datagen.GenRandomAddress()

	initialRatio := sdkmath.NewInt(50)
	dp := types.DefaultParams()
	dp.ScoreRatioBtcByBaby = initialRatio
	err := k.SetParams(ctx, dp)
	require.NoError(t, err)

	// Attacker: 5000 sats, 50000 baby -> score = min(5000,
50000/50) = 1000
	activeSats1 := sdkmath.NewInt(5000)
	activeBaby1 := sdkmath.NewInt(50000)
	err = k.costakerModifiedActiveAmounts(ctx, attacker,
activeSats1, activeBaby1)
	require.NoError(t, err)

	// Victim: 10000 sats, 100000 baby -> score = min(10000,
100000/50) = 2000
	activeSats2 := sdkmath.NewInt(10000)
	activeBaby2 := sdkmath.NewInt(100000)
	err = k.costakerModifiedActiveAmounts(ctx, victim, activeSats2,
activeBaby2)
	require.NoError(t, err)

	// Verify total score: 3000
	currentRwd, err := k.GetCurrentRewards(ctx)
	require.NoError(t, err)
	require.Equal(t, sdkmath.NewInt(3000), currentRwd.TotalScore)

	// Deposit 30,000 tokens as rewards
	totalDeposited :=
sdk.NewCoins(sdk.NewCoin(appparams.DefaultBondDenom,
sdkmath.NewInt(30000)))
        err = k.AddRewardsForCostakers(ctx, totalDeposited)
        require.NoError(t, err)

        // Finalize the period so rewards become claimable
        periodBeforeUpdate, err := k.IncrementRewardsPeriod(ctx)
        require.NoError(t, err)

        // BEFORE UPDATE: Calculate total claimable rewards
        attackerRewardsBefore, err := k.CalculateCostakerRewards(ctx,
attacker, periodBeforeUpdate)
        require.NoError(t, err)
        victimRewardsBefore, err := k.CalculateCostakerRewards(ctx,
victim, periodBeforeUpdate)
        require.NoError(t, err)

        totalClaimableBefore :=
attackerRewardsBefore.Add(victimRewardsBefore...)

        t.Logf("\n=== BEFORE GOVERNANCE UPDATE ===")
        t.Logf("Total deposited rewards:    %s", totalDeposited)
        t.Logf("Attacker claimable:         %s (score: 1000)",
attackerRewardsBefore)
        t.Logf("Victim claimable:           %s (score: 2000)",
victimRewardsBefore)
        t.Logf("Total claimable:            %s", totalClaimableBefore)
        t.Logf("Invariant check:            %s <= %s  ✓",
totalClaimableBefore, totalDeposited)

        // Verify invariant BEFORE update: claimable <= deposited
        require.True(t, totalClaimableBefore.IsAllLTE(totalDeposited),
                "BEFORE update: total claimable (%s) should be <=
deposited (%s)",
                totalClaimableBefore, totalDeposited)

        // GOVERNANCE UPDATES PARAMETER
        // This changes the ratio from 50 to 25, doubling scores for
these costakers
        newRatio := sdkmath.NewInt(25)
        t.Logf("\n=== GOVERNANCE PARAMETER UPDATE ===")
        t.Logf("ScoreRatioBtcByBaby: 50 -> 25")

        err = k.UpdateAllCostakersScore(ctx, newRatio)
        require.NoError(t, err)

        // Verify scores doubled
        attackerTrackerAfter, err := k.GetCostakerRewards(ctx,
attacker)
        require.NoError(t, err)
        victimTrackerAfter, err := k.GetCostakerRewards(ctx, victim)
        require.NoError(t, err)

        require.Equal(t, sdkmath.NewInt(2000),
attackerTrackerAfter.TotalScore, "Attacker score should double")
        require.Equal(t, sdkmath.NewInt(4000),
victimTrackerAfter.TotalScore, "Victim score should double")

        t.Logf("Attacker NEW score: %s (doubled from 1000)",
attackerTrackerAfter.TotalScore)
```

```
        t.Logf("Victim NEW score:   %s (doubled from 2000)",
victimTrackerAfter.TotalScore)

        // CHECK: What happened to StartPeriodCumulativeReward?
        attackerTrackerBefore, err := k.GetCostakerRewards(ctx,
attacker)
        require.NoError(t, err)
        victimTrackerBefore, err := k.GetCostakerRewards(ctx, victim)
        require.NoError(t, err)

        t.Logf("\n=== TRACKER STATE CHECK ===")
        t.Logf("Attacker StartPeriodCumulativeReward: %d",
attackerTrackerBefore.StartPeriodCumulativeReward)
        t.Logf("Victim StartPeriodCumulativeReward:   %d",
victimTrackerBefore.StartPeriodCumulativeReward)
        t.Logf("Period that was finalized:            %d",
periodBeforeUpdate)

        // AFTER UPDATE: Calculate total claimable rewards
        // The key bug: we calculate from the ORIGINAL period
(periodBeforeUpdate)
        // but with the NEW inflated scores!
        attackerRewardsAfter, err := k.CalculateCostakerRewards(ctx,
attacker, periodBeforeUpdate)
        require.NoError(t, err)
        victimRewardsAfter, err := k.CalculateCostakerRewards(ctx,
victim, periodBeforeUpdate)
        require.NoError(t, err)

        totalClaimableAfter :=
attackerRewardsAfter.Add(victimRewardsAfter...)

        t.Logf("\n=== AFTER GOVERNANCE UPDATE ===")
        t.Logf("Total deposited rewards:    %s (UNCHANGED)",
totalDeposited)
        t.Logf("Attacker claimable:         %s (score: 2000)",
attackerRewardsAfter)
        t.Logf("Victim claimable:           %s (score: 4000)",
victimRewardsAfter)
        t.Logf("Total claimable:            %s", totalClaimableAfter)
        t.Logf("Invariant check:            %s <= %s ✗ VIOLATED!",
totalClaimableAfter, totalDeposited)

        // Verify invariant is broken after update
        require.False(t, totalClaimableAfter.IsAllLTE(totalDeposited),
                "BUG DEMONSTRATED: total claimable (%s) EXCEEDS
deposited (%s)!",
                totalClaimableAfter, totalDeposited)
        require.Equal(t, "20000ubbn", attackerRewardsAfter.String(),
                "Attacker can claim 20000 (was 10000) - DOUBLED")
        require.Equal(t, "40000ubbn", victimRewardsAfter.String(),
                "Victim can claim 40000 (was 20000) - DOUBLED")
        require.Equal(t, "60000ubbn", totalClaimableAfter.String(),
                "Total claimable is 60000 but only 30000 was
deposited!")

        // Calculate the excess
        excessClaimable := totalClaimableAfter.Sub(totalDeposited...)
        t.Logf("\n=== VULNERABILITY SUMMARY ===")
        t.Logf("Deposited:        %s", totalDeposited)
        t.Logf("Claimable before: %s (invariant holds)",
totalClaimableBefore)
        t.Logf("Claimable after:  %s (invariant BROKEN)",
totalClaimableAfter)
        t.Logf("EXCESS:           %s (200%% of deposits!)",
excessClaimable)
        t.Logf("")
        t.Logf("ROOT CAUSE: UpdateAllCostakersScore updates scores but
does NOT")
        t.Logf("update StartPeriodCumulativeReward, allowing costakers
to claim")
        t.Logf("historical rewards retroactively with their NEW
inflated scores.")
        t.Logf("")
        t.Logf("IMPACT: First withdrawers drain the module, leaving it
insolvent.")
        t.Logf("Remaining costakers cannot claim their legitimate
rewards.")
}
```

V4-CORE-002

Integer overflow in derivation path parsing

Status
Solved

Resolution
Fixed

Risk
Low

Impact
Medium

Likelihood
Low

Location

`covenant-emulator/covenant-signer/keyutils/bip32deriv.go`

Description

The covenant-signer module's `DeriveChildKey` function is susceptible to an integer overflow when parsing hardened derivation paths. The `parseHardened` function does not validate if the parsed number, `numAsUint32`, exceeds the maximum possible value before adding the `hdkeychain.HardenedKeyStart` constant.

```go
func parseHardened(elem string) (uint32, error) {
    //...

    parsedNum, err := strconv.ParseUint(number, 10, 32)
    if err != nil {
        return 0, fmt.Errorf("invalid hardened element")
    }

    numAsUint32 := uint32(parsedNum)

    // todo check for overflow
    return hdkeychain.HardenedKeyStart + numAsUint32, nil
}
```

If a user provides a hardened path element where `numAsUint32` is a value greater than `0x7FFFFFFF`, the addition of `hdkeychain.HardenedKeyStart` (which is equal to `0x80000000`) will cause an overflow. This leads to an incorrect child key being derived. The comment `// todo check for overflow` indicates that the developers were aware of this missing validation, but it has not been implemented.

Coinspect considers the likelihood as low as this is intended for use by a trusted operator, not a publicly exposed API. The operator has full control over the inputs and is not expected to deliberately cause an overflow. The impact is medium because if the operator mistakenly inputs an incorrect path, the derived key will be incorrect.

Recommendation

Implement a check in the `parseHardened` function that validates that the parsed number does not exceed the maximum allowed value before adding `hdkeychain.HardenedKeyStart`.

Status

Fixed at PR #156.

V4-CORE-003

Finality provider does not check for error on writing public randomness

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
–

Location

`eotsmanager/randgenerator/randgenerator.go`

Description

The `GenerateRandomness` method of the `eotsmanager` of a finality provider does not check the `err` return value of the `Write` call into the `digest`:

```
// calculate the random hash with iteration count
digest := hmac.New(sha256.New, key)
digest.Write(append(append(sdk.Uint64ToBigEndian(height), chainID...),
sdk.Uint64ToBigEndian(iteration)...))
randPre := digest.Sum(nil)
```

While the `hmac.New` `io.Writer` does not err-out in any situation with the current implementation, the `io.Writer` contract does specify a possible `err` return value that *must* be checked to know if the whole buffer has been written to.

If internal details of the implementation change, an error will be silently ignored by the finality provider, potentially resulting in repeated public randomness.

Recommendation

Check for `err` on the `Write` call.

Status

Fixed at PR #724.

V4-CORE-004

Finality provider can generate invalid 0×00 private keys

Status

Solved

✓

Resolution

Fixed

Risk

Low

🟥⬜⬜

Impact

Medium

Likelihood

Low

Location

```
finality-provider/
```

Description

The `btcd` library used by Babylon's finality provider has a bug which can make it sign messages with invalid keys.

`btcd`'s Sign() function will sign a message in compliance with BIP-340, but it assumes that the `privKeyScalar` is below the curve order `n`.

```
// Step 3.
//
// Fail if d = 0 or d >= n
if privKeyScalar.IsZero() {
    str := "private key is zero"
    return nil, signatureError(ecdsa_schnorr.ErrPrivateKeyIsZero, str)
}
```

Nevertheless, this assumption is wrong. When calling `PrivKeyFromBytes` the slice is *not* checked for compliance on the curve:

```
// PrivKeyFromBytes returns a private and public key for `curve' based
on the
// private key passed as an argument as a byte slice.
func PrivKeyFromBytes(pk []byte) (*PrivateKey, *PublicKey) {
        privKey := secp.PrivKeyFromBytes(pk)

        return privKey, privKey.PubKey()
}
```

This happens even though the inner function used, `secp.PrivKeyFromBytes` explicitly states that this is the callers responsibility:

```
//
// ...
// WARNING: This means passing a slice with more than 32 bytes is
truncated and
// that truncated value is reduced modulo N.  Further, 0 is not a valid
private
// key.  It is up to the caller to provide a value in the appropriate
range of
// [1, N-1].  Failure to do so will either result in an invalid private
key or
// potentially weak private keys that have bias that could be
exploited.
// ...
func PrivKeyFromBytes(privKeyBytes []byte) *PrivateKey { ... }
```

All in all, this means that `btcd` is passing responsibility to *its* callers to check the private key bytes before converting it to the `PrivateKey` type.

While the risk is negligible because:

- The risk of generating random numbers outside of `secp256k1` order is exceedingly small
- The key-derivation depends on Cosmo's key manager, which might be generating valid private keys

Coinspect still recommends making sure they keys are in the order before using them to sign.

Recommendation

Check that the private key fits in the order of `secp256k1` before using it to sign.

Status

Fixed at PR #725.

V4-CORE-005

Finality provider can double sign due to race condition

Status
Solved

✓

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

`finality-provider/eotsmanager/localmanager.go`

Description

Concurrent calls to `SignEOTS` or `SignBatchEOTS` can sign the same height with different messages using the same nonce, bypassing the double sign protection of the finality provider software and causing the operator to sign and broadcast two messages at the same height. This is because `SignEOTS` and `SignBatchEOTS` perform do not have atomicity protection

When two RPC requests arrive concurrently for the same height:

1. Both goroutines read existing signatures
2. Both find no existing signature for height `h`
3. Both generate EOTS signatures for height `h` with the same nonce
4. Both write their signatures

This renders the double signing protection for finality providers ineffective.

The vulnerability exists in both methods in `eotsmanager/localmanager.go`, as an example `SignEOTS` is shown with comments detailing the problematic code:

```
func (lm *LocalEOTSManager) SignEOTS(eotsPk []byte, chainID []byte, msg
[]byte, height uint64) (*btcec.ModNScalar, error) {
    // Line 238: Read phase - NOT ATOMIC
    record, found, err := lm.es.GetSignRecord(eotsPk, chainID, height)

    // Line 244-280: Check phase - uses potentially stale data
    if found {
        if bytes.Equal(msg, record.Msg) {
            // Reuse existing signature
            return &s, nil
        }
        return nil, eotstypes.ErrDoubleSign
    }

    // Line 299: Sign phase - IRREVERSIBLE
    signedBytes, err := eots.Sign(privKey, privRand, msg)

    // Line 305: Write phase - too late to prevent race
    if err := lm.es.SaveSignRecord(height, chainID, msg, eotsPk, b[:]);
err != nil
}
```

The database's duplicate checks in `SaveSignRecord` and `SaveSignRecordsBatch` occur after signatures are already generated, making them ineffective for preventing nonce reuse.

The likelihood of this issue is considered `low` both because the finality provider would have to first experience some kind of glitch that would send two requests for the same height and because the race window is very small: to observe the issue in tests Coinspect introduces a random `time.Sleep` delay in the vulnerable logic, which increased the window and allowed the bug to be reliably observed in tests. Nevertheless, its impact is `high` because if triggered the consequences are critical for the finality provider.

Recommendation

Add a mutex to serialize the entire read-check-sign-write sequence in both signing methods.

Status

Fixed at PR #726.

V4-CORE-006

Attacker can cause total BABY stake to be miscounted

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`babylon/x/costaking`

Description

An attacker can cause the total BABY stake to be miscounted by the `x/costaking` module, as a full undelegation followed by a new delegation to the same validator in the same block leads to a loss of the newly staked BABY tokens in the `x/costaking` module's accounting. This impacts both the affected costaker by undercounting his rewards, and the overall system by corrupting the `TotalScore` used for global reward calculation.

The regular flow of the staking hooks and the staking cache is as follows:

- Regular Flow (Delegation Modification): When an existing delegation is modified (e.g., more tokens are added or some are undelegated), the staking module calls `BeforeDelegationSharesModified`. This hook correctly uses the cache to store the previous amount. The subsequent `AfterDelegationModified` hook then retrieves this cached amount to

calculate the delta by subtracting the cached amount from the new amount: `Delta = NewAmount - CachedOldAmount`.
- Special Case Flow (New Delegation): For a new delegation where no prior delegation existed, the staking module calls `BeforeDelegationCreated` which does nothing on `x/costaking`. The `AfterDelegationModified` hook is then called, but since the cache is empty, `CachedOldAmount` defaults to zero (as per `GetStakedAmount` logic), and the full amount of the new delegation is correctly registered as the `Delta`.

The issue originates from the fact that creating a new delegation triggers the `BeforeDelegationCreated` hook instead of `BeforeDelegationSharesModified` hook and expects that the cache is empty to work properly. The issue can be triggered by executing the following actions:

1. A staker has tokens delegated to a validator (`Delegation` object exists).
2. Full undelegation deletes the `Delegation` object, and the `BeforeDelegationSharesModified` hook caches the original, non-zero amount. The `ActiveBaby` amount is correctly set to zero when the `AfterDelegationModified` hook is called.
3. The staker delegates again to the same validator in the same block. Since the delegation object was deleted, the `Delegate` function executes the logic for a new delegation. This triggers the `BeforeDelegationCreated` hook, which unlike `BeforeDelegationSharesModified`, does not cache the current delegation amount (which is zero at this point) and the cache is not modified, keeping the same value as before unbonding.
4. The `AfterDelegationModified` hook is called after the delegation is created. This hook calculates the change in BABY tokens to update `ActiveBaby` and uses the incorrect cached value.

```
ActiveBaby = PreviousBaby + (NewDelegationAmount - DelegationTokensBefore)
```

- `PreviousBaby` is the costaker's ActiveBaby after step (2), which is zero.

- `DelegationTokensBefore` is the cached amount from step (2) (set by `BeforeDelegationSharesModified`), which is the initial delegation amount before the full undelegation.

- `NewDelegationAmount` is the amount of the new delegation (step 3).

The new `ActiveBaby` calculation becomes:

```
ActiveBaby = 0 + (NewDelegationAmount - InitialDelegationAmount)
```

If `NewDelegationAmount=InitialDelegationAmount`, the resulting new `ActiveBaby` is zero, meaning the tokens from the new delegation are effectively lost to the costaking system and do not count towards rewards. If `NewDelegationAmount < InitialDelegationAmount`, the transaction fails because the new `ActiveBaby` would be negative. If `NewDelegationAmount > InitialDelegationAmount`, the new `ActiveBaby` is only the difference, undercounting the staked amount.

This miscount directly affects the individual staker's rewards calculation and globally corrupts the `CurrentRewards.TotalScore`, as it is derived from the sum of all stakers' active tokens, leading to incorrect global reward distribution.

Recommendation

Clear the staking cache for the specific delegator/validator pair after the delta has been calculated.

Status

Fixed at PR #1792

V4-CORE-007

Attacker can steal all rewards by restaking BABY tokens

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`babylon/x/costaking/keeper/hooks_staking.go`

Description

An attacker can multiply their costaking rewards by an arbitrary factor by repeatedly delegating and completely undelegating the same BABY tokens. The attack exploits the fact that `BeforeDelegationRemoved` just returns `nil` and does not update `ActiveBaby` accounting when a delegation is completely removed. `BeforeDelegationRemoved` is called by the underlying Cosmos SDK when the delegation is fully unbonded from a validator.

The `x/costaking` module tracks staked BABY tokens through the `ActiveBaby` field in the costaker rewards tracker. This field should represent the total amount of BABY currently staked. The module implements `AfterDelegationModified` to handle changes in delegation amounts, but fails to implement `BeforeDelegationRemoved`:

```
func (h HookStaking) BeforeDelegationRemoved(ctx context.Context,
delAddr sdk.AccAddress, valAddr sdk.ValAddress) error {
    return nil
}
```

The Cosmos SDK staking module calls different hooks depending on the type of undelegation:

- **Partial undelegation** (shares > 0 remain): Calls `AfterDelegationModified`
- **Complete undelegation** (shares = 0, delegation deleted): Calls `BeforeDelegationRemoved`

Since x/costaking only implements the first hook, complete undelegations never update `ActiveBaby`.

This allows the following attack:

1. Attacker delegates N BABY to validator V1
    - `AfterDelegationModified` adds N to ActiveBaby
    - ActiveBaby = N
2. Attacker undelegates ALL N BABY from V1
    - Cosmos SDK calls `BeforeDelegationRemoved` (not `AfterDelegationModified`)
    - Hook does nothing - returns nil
    - ActiveBaby remains N (should be 0)
3. Attacker delegates same N BABY to validator V2
    - `AfterDelegationModified` adds N to ActiveBaby again
    - ActiveBaby = N + N = 2N
4. Repeat steps 2-3 with validators V3, V4, ..., VM
    - Each complete undelegate does nothing
    - Each new delegation adds N
    - Final ActiveBaby = N * M

The attacker now receives M times their legitimate share of costaking rewards while only having N BABY tokens actually staked.

The root cause is the missing implementation of `BeforeDelegationRemoved`. The hook exists (line 100-102) but contains no logic.

Recommendation

Implement `BeforeDelegationRemoved` to subtract the delegation amount from ActiveBaby. Use the cached amount from `BeforeDelegationSharesModified`:

Status

Fixed at PR #1789

V4-CORE-008

Potential misaccounting in RemoveBondedSats due to underflow safeguard

Status
Solved

✓

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
–

Location

`x/finality/types/power_table.go`

Description

The `RemoveBondedSats` function includes a safeguard to prevent an underflow. If the amount of satoshis to be removed (`sats`) is greater than the finality provider's `TotalBondedSat`, the function caps the removal amount to `v.TotalBondedSat`, effectively setting the balance to zero. While this prevents a potential underflow, it silently handles a state that should likely be an error, which could lead to a misaccounting of bonded sats.

An adversary may not be able to abuse this directly, but it creates a scenario where the system might account for more bonded `sats` being removed than what was actually deducted from the finality provider's total.

```go
func (v *FinalityProviderDistInfo) RemoveBondedSats(sats uint64) {
	// safeguard against underflow in total bonded satoshis
	if v.TotalBondedSat < sats {
		sats = v.TotalBondedSat

	}
	v.TotalBondedSat -= sats
}
```

Coinspect performed tests and, under current conditions, this block is not entered. However, even though this behavior was found not presently exploitable, it introduces a potential for future bugs where the finality provider's accounted bonded sats could diverge from the actual value tracked in its state.

Recommendation

Consider modifying the `RemoveBondedSats` function to error when `sats` is greater than `v.TotalBondedSat`.

Status

Fixed at PR #1813.

V4-WEB-001

Adversaries can abuse Chainalysis API usage quota due to missing rate limiting

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
Medium

Likelihood
Medium

Location

```
staking-api-service/internal/shared/api/routes.go
staking-api-service/internal/v2/api/handlers/address.go
```

Description

The `/address/screening` endpoint does not implement any rate-limiting mechanism. This allows an adversary to repeatedly call the endpoint, which triggers a request to the external Chainalysis API for each call, to cause a denial of service by exhausting the API quota or to inflict financial damage by arbitrarily increasing the billing costs associated with the Chainalysis' service.

As shown below, the route is registered without any specific middleware for rate limiting:

```
if a.cfg.AddressScreeningConfig != nil &&
a.cfg.AddressScreeningConfig.Enabled {
    r.Get("/address/screening",
registerHandler(handlers.V2Handler.AddressScreening))
}
```

The `AddressScreening` handler directly calls the `AssessAddress` service, which in turn calls the Chainalysis API, without any checks to prevent abuse.

Recommendation

Implement a strict rate-limiting strategy for the `/address/screening` endpoint.

Status

Fixed. Babylon Labs stated that they have alerts to prevent this kind of attacks.

V4-WEB-002

Potentially inaccurate unbonding statistics

Status
Solved

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
-

Location

`internal/v1/service/unbonding.go`

Description

The `UnbondDelegation` function updates unbonding statistics immediately after a valid unbonding request reaches the API server. Apart from the behavior documented in the code (see below), it is problematic because it only reflects requests handled by a single server instance, ignoring unbondings initiated through others.

The code notes the limitation—that statistics are updated without waiting for Bitcoin confirmation—but it does not account for unbonding requests processed by different servers.

```
// This is a temporary solution to keep phase-1 stats up to date with
the
// unbonding triggered by the staker. Ideally, the stats should only be
// calculated when the actual unbonding tx is confirmed on BTC. But API
service
// does not have visibility into this. and considering this is a
```

```
temporary
// solution in which the whole phase-1 stats will be removed right
after phase-2
// is launched, we will process the stats calculation here based on the
assumption
// that all requested unbonding will be processed eventually.
```

As a result, the system produces inconsistent and inaccurate network-wide statistics, since each server has only a partial view of unbonding activity. Although acknowledged as a temporary workaround, this design risks presenting a misleading picture of staking metrics.

Recommendation

Unbonding statistics should be updated only after the unbonding transaction is confirmed on the Bitcoin blockchain and indexed by Babylon. This ensures a single, authoritative source of truth for all unbonding events, independent of which API server processed the initial request.

Status

Acknowledged. Babylon Labs stated they are aware of this limitation and take it into account when interpreting their statistics.

V4-WEB-003

Lack of rate limiting and ownership verification in terms acceptance endpoint

Status	Risk
Solved	**Low**
✓	🟥⬜⬜
Resolution: **Fixed**	Impact: **Low**
	Likelihood: **Low**

Location

`staking-api-service/internal/shared/api/handlers/handler/terms.go`

Description

The `LogTermsAcceptance` endpoint allows an adversary to pollute the database and accept terms on behalf of arbitrary users. An adversary can abuse this problem to fill the database with fraudulent terms acceptance objects, potentially leading to a denial-of-service condition, and create false records for legitimate users without their consent.

The vulnerability stems from two main issues:

1. **No Rate Limiting:** The endpoint does not enforce any rate-limiting, allowing an attacker to submit an unlimited number of requests, filling the database with junk entries.
2. **Missing Ownership Verification:** The endpoint accepts a public key and an address without verifying that they are related or that the requester possesses the corresponding private key. The

`parseTermsAcceptanceLoggingRequest` function only checks for format validity, not for the cryptographic link between the address and the public key. This allows anyone to submit a request for any address, effectively accepting terms on behalf of a third party.

Recommendation

Implement a rate-limiting mechanism such as captcha.

Enforce proof of key possession by requiring the client to sign a predefined message with the private key corresponding to the provided public key.

Make sure the provided address is indeed derived from the public key.

Status

Fixed in PR #462 and PR #429. The term acceptance endpoint has been removed.

V4-WEB-004

Denial of Service due to unbounded mongoDb query

Status
Solved

✓

Resolution
Fixed

Risk
High

🟥🟥🟥

Impact
High

Likelihood
High

Location

`internal/indexer/db/client/finality_provider.go`

Description

The `GetFinalityProviders` function in the `finality_provider.go` client and its helper `pkg.FetchAll`, fetch all finality provider documents from the database into memory at once. An adversary can abuse this problem to cause a denial of service via memory exhaustion on all the API servers.

This function is called by the service layer, which is exposed via the following API endpoints:

- `GET /v2/finality-providers`
- `GET /v1/finality-providers`

Specifically, an attacker can trigger the unbounded query by calling the V2 endpoint that does not support pagination. This forces the server to load all

FP documents into memory, potentially leading to an out-of-memory error and crashing the server.

```
func (indexerdbclient *IndexerDatabase) GetFinalityProviders(
        ctx context.Context,
) ([]*indexerdbmodel.IndexerFinalityProviderDetails, error) {
        client := indexerdbclient.Client.Database(
                indexerdbclient.DbName,
        ).Collection(indexerdbmodel.FinalityProviderDetailsCollection)

        // Fetch all finality providers
        filter := bson.M{}

        return
pkg.FetchAll[*indexerdbmodel.IndexerFinalityProviderDetails](ctx,
client, filter)
}
```

```
func FetchAll[T any](ctx context.Context, collection *mongo.Collection,
filter bson.M) ([]T, error) {
        cursor, err := collection.Find(ctx, filter)
        if err != nil {
                return nil, err
        }
        defer cursor.Close(ctx)

        var result []T
        for cursor.Next(ctx) {
                var doc T

                err = cursor.Decode(&doc)
                if err != nil {
                        return nil, err
                }

                result = append(result, doc)
        }

        if cursor.Err() != nil {
                return nil, cursor.Err()
        }

        return result, nil
}
```

Creating new Finality Providers (FPs) is permissionless, and its cost is relatively negligible. Although the size of a single FP document is small, an attacker can create thousands of them throughout an extended time window.

The following constant definition corresponds to the size cap enforced by the `validateDescription` function in the Babylon node when creating and editing new Finality Providers. Along with the rest of the fields, this adds approximately 4kb.

```
const (
        // TODO: Why can't we just have one string description which
can be JSON by convention
        MaxMonikerLength         = 70
        MaxIdentityLength        = 3000
        MaxWebsiteLength         = 140
        MaxSecurityContactLength = 140
        MaxDetailsLength         = 280
)
```

To generate a single gigabyte of data via new Finality Providers, considering an average block time of 10 seconds and a maximum block size of 22mb, an attacker would need a little less than 8 minutes. This would need to fill blocks up to 300M gas limit. Assuming that filling the block and consuming all the gas limit are equivalent, the amount of baby required to fill the whole block can be calculated as follows: 300M (gasLimit) * 0.002 (ubbn / gas) = 600000ubbn = 0.6baby ~ 0.03USD (in current prices).

At an average cost of 0.6 BABY per block, generating a gigabyte can be achieved in 48 blocks and would cost approximately 28.8 BABY.

Recommendation

Implement pagination for the `GetFinalityProviders` database query and the corresponding API endpoints.

Review the rest of the `pkg.FetchAll()` invocations and consider only using it with paginated queries.

Status

Fixed in PR #428. The `FetchAll` logic has been replaced by a more efficient grouping on the database and pagination logic has been implemented.

V4-WEB-005

Usage of a fixed gas price might lead to stuck transactions

Status

Solved

✓

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
babylon-toolkit/services/simple-staking/src/ui/common/config/index.ts
```

Description

The application uses a fixed gas price, `BBN_GAS_PRICE`, which is set from the `process.env.NEXT_PUBLIC_BBN_GAS_PRICE` environment variable. While using a fixed `gasPrice` is a common practice for Cosmos-based dapps, nodes often expect a minimum `gasPrice` to include transactions in a block. If the configured gas price is lower than the minimum expected by the nodes, transactions may not be processed, leading to them being stuck.

The current implementation reads the gas price from an environment variable and falls back to a default value if it's not set. This does not account for potential changes in the minimum gas price required by the network's nodes, which can fluctuate. This scenario is aggravated by the fact that there is no transaction retry mechanism in place for transactions that fail due to insufficient gas.

Recommendation

Consider implementing a mechanism to allow the dapp to query the minimum gas price expected by nodes.

Additionally, a transaction retry mechanism could be implemented to handle cases where transactions fail due to insufficient fees.

Status

Acknowledged. Babylon Labs stated the gas price is in-line with expectations.

V4-WEB-006

Malformed staking expansion transaction on API failure

Status
Solved

Resolution
Fixed

Risk
Low

Impact
Low

Likelihood
Low

Location

```
babylon-toolkit/services/simple-
staking/src/ui/common/hooks/services/useStakingExpansionService.ts
```

Description

In the `stakeDelegationExpansion` function, the `previousStakingTxHex` variable is incorrectly initialized with the `stakingTxHex` of the current (new) delegation. The logic then attempts to fetch the original delegation to get the correct, previous transaction hex. However, if this API call fails or returns no data, the function does not handle the error and instead proceeds silently using the incorrect transaction hex.

An adversary cannot directly trigger this, but any intermittent API failure could cause the application to try and build an expansion transaction using itself as the "previous" transaction. This would lead to a failed transaction on the client side or the submission of a malformed transaction, creating a confusing user experience and potentially an invalid state.

```
// Get the original delegation data if this is an expansion
let previousStakingTxHex = delegation.stakingTxHex;
let previousStakingInput = {
    finalityProviderPksNoCoordHex:
delegation.finalityProviderBtcPksHex,
    stakingAmountSat: delegation.stakingAmount,
    stakingTimelock: delegation.stakingTimelock,
};

// If this is an expansion, fetch the original delegation data
if (delegation.previousStakingTxHashHex) {
    const originalDelegation = await getDelegationV2(
    delegation.previousStakingTxHashHex,
    );
    if (originalDelegation) {
    previousStakingTxHex = originalDelegation.stakingTxHex;
    previousStakingInput = {
        finalityProviderPksNoCoordHex:
        originalDelegation.finalityProviderBtcPksHex,
        stakingAmountSat: originalDelegation.stakingAmount,
        stakingTimelock: originalDelegation.stakingTimelock,
    };
    }
}
```

Recommendation

Implement robust error handling for the `getDelegationV2` API call within this function. If fetching the original delegation fails, the function should throw an error to halt execution.

Status

Fixed in PR #525.

V4-WEB-007

Geo-block can be bypassed

Status
Solved

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
–

Location

```
babylon-toolkit/services/simple-
staking/src/ui/common/services/healthCheckService.ts
```

Description

The current health check mechanism is implemented on the client-side in the `getHealthCheck` function. This function relies on the server's response to determine if the user's access should be restricted (e.g., due to geo-blocking). Someone trying to bypass the geo-block can intercept and modify the server's response, effectively bypassing the check and gaining access to the service from a restricted IP address or while using a VPN.

The client-side code trusts the server's response without any server-side enforcement, rendering the security control ineffective against a determined attacker. The function at `healthCheckService.ts` attempts to handle a geo-blocking scenario by checking for a specific error status, but it assumes a successful response indicates the user is cleared for access.

```
export const getHealthCheck = async (): Promise<HealthCheckResult> => {
  try {
    const healthCheckAPIResponse = await fetchHealthCheck();

    return {
      status: HealthCheckStatus.Normal,
      message: healthCheckAPIResponse.data,
    };
  } catch (error: any) {
    if (isError451(error.cause)) {
      throw new ClientError(ERROR_CODES.GEO_BLOCK, GEO_BLOCK_MESSAGE, {
        cause: error.cause,
      });
    }
    throw error;
  }
};
```

Recommendation

Do not rely on the client-side logic to enforce geo-blocking. Instead, the API server should deny service to requests originating from blocked IPs or VPNs directly.

Status

Acknowledged by Babylon Labs team.

V4-WEB-008

Redundant transaction confirmation logic

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
-

Location

```
babylon-toolkit/services/simple-
staking/src/ui/common/hooks/services/useRewardsService.ts
```

Description

The `claimRewards` function in the `useRewardsService` hook contains a `retry` mechanism that attempts to confirm a successful reward withdrawal by polling for a change in the user's balance. This logic is entirely redundant and has no effect on the application's behavior. An adversary cannot abuse this problem, but it introduces unnecessary complexity and can mislead developers into thinking the application has a robust confirmation mechanism when it does not.

The underlying `sendBbnTx` function already ensures transaction finality. It utilizes `signingStargateClient.broadcastTx`, which waits for the transaction to be included in a block and checks the transaction's `code`. If the code is non-zero (indicating an on-chain failure), it throws an error, which is correctly handled by the existing `try...catch` block. The subsequent `retry` logic to check for a balance change is therefore superfluous, as the transaction has already been confirmed at this point.

```
await refetchRewardBalance();
const initialBalance = balanceQuery.data || 0;
await retry(
    () => balanceQuery.refetch().then((res) => res.data),
    (value) => value !== initialBalance,
    ONE_SECOND,
    MAX_RETRY_ATTEMPTS,
);
```

This unnecessary logic complicates code maintenance and can lead to confusion about how transaction success is determined. The result of the retry call is not checked, so it has no impact on the function's execution path, making it dead code.

Recommendation

Remove the entire `retry` function from the `claimRewards` function.

Status

Fixed in PR #486.

V4-WEB-009

Ineffective transaction hash validation in staking expansion

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
–

Location

```
babylon-toolkit/services/simple-
staking/src/ui/common/hooks/services/useStakingExpansionService.ts
```

Description

The `createExpansionEOI` function includes a validation check that is logically flawed and can never fail. The `stakingTxHashHex` variable is assigned the value of `stakingTxHash` on line 217, and the subsequent `if` statement compares these two identical values. This renders the check useless.

This dead code provides a false sense of security. While not directly exploitable, it represents a logical error that could mask other underlying bugs or lead to incorrect behavior if the surrounding code is refactored under the assumption that this validation is functional.

```
const stakingTxHashHex = stakingTxHash; // Use the hash from the
expansion result
// BBN transaction sent successfully
```

```
if (stakingTxHashHex !== stakingTxHash) {
    const clientError = new ClientError(
    ERROR_CODES.VALIDATION_ERROR,
    `Staking expansion transaction hash mismatch, expected
${stakingTxHash} but got ${stakingTxHashHex}`,
    );
    throw clientError;
}
```

Recommendation

Remove the redundant `stakingTxHashHex` variable and the associated conditional check that can never fail.

Status

Fixed at PR #408.

V4-WEB-010

Stale tipHeight can lead to transaction failure or invalid staking transaction

Status
Solved

✓

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
-

Location

```
babylon-toolkit/services/simple-
staking/src/ui/common/hooks/services/useTransactionService.ts
```

Description

A potential race condition exists in the `useTransactionService` hook related to how the Bitcoin `tipHeight` is handled, specifically for the `createDelegationEoi` function. This would lead to the creation of unintended delegation transactions, which could result in an invalid transaction.

The `tipHeight` is calculated and stored when the component initializes. If there is a delay between when this height is determined and when a user initiates a staking transaction, the `tipHeight` can become stale. If staking parameters on the Babylon chain change during this window, a transaction created with the outdated `tipHeight` may be rejected by the network or result in a malformed staking transaction. This could cause the transaction to be invalid or behave unexpectedly.

Recommendation

Refetch the latest `tipHeight` from the `useBbnQuery` hook immediately before it is used to construct any transaction.

Status

Fixed at PR #407.

6. Disclaimer

The contents of this report are provided "as is" without warranty of any kind. Coinspect is not responsible for any consequences of using the information contained herein.

This report represents a point-in-time and time-boxed evaluation conducted within a specific timeframe and scope agreed upon with the client. The assessment's findings and recommendations are based on the information, source code, and systems access provided by the client during the review period.

The assessment's findings should not be considered an exhaustive list of all potential security issues. This report does not cover out-of-scope components that may interact with the analyzed system, nor does it assess the operational security of the organization that developed and deployed the system.

This report does not imply ongoing security monitoring or guaranteeing the current security status of the assessed system. Due to the dynamic nature of information security threats, new vulnerabilities may emerge after the assessment period.

This report should not be considered an endorsement or disapproval of any project or team. It does not provide investment advice and should not be used to make investment decisions.