coinspect

You build, we defend.

babylon

Source Code Audit

Phase 2

March 2025

Babylon Phase 2

Source Code Audit

Version: v250324 Prepared for: Babylon March 2025

Security Assessment

1. Executive Summary

2. Summary of Findings

2.3 Solved issues & recommendations

3. Scope

3.1 Review of fixes

4. Assessment

4.1 Security assumptions

4.2 Mitigating supply chain and infrastructure risks

5. Detailed Findings

BP2-001 - Minority of stakers can cause inconsistent finalization

BP2-002 - Recursion in AnteHandler allows for potentially arbitrary stack-depth

BP2-003 - Attacker can avoid checkpointing state by sending more than one message

BP2-004 - Attacker can spam mempool with Bitcoin related messages

BP2-005 - Attacker can avoid checkpointing state by using WASM contracts

BP2-006 - Default transaction fee cap may delay Bitcoin transaction confirmation

BP2-007 - User can waste time and funds due to generation of non-standard transactions

BP2-008 - Unsupported TLS connection to bitcoind nodes

BP2-009 - Attacker can avoid checkpointing with embedded messages in WASM contracts

BP2-010 - Attacker can stall chain with WASM contracts

BP2-011 - Governance can approve messages that bypass checkpointing

BP2-012 - Submitter will crash due to unhandled null return value

BP2-013 - Adversary can force Babylon node to waste resources by reprocessing known blocks

BP2-014 - Vigilante Reporter will eventually crash due to unbounded map growth

BP2-015 - Transaction execution at risk due to decreased feerate

BP2-016 - Bitcoin transactions not fully validated for standardization in Babylon node

BP2-017 - Slashing a validator will cause the node to crash

BP2-018 - Wrong calculation of default transaction fee bump prevents bumping transactions

BP2-019 - Replacement transactions might not be accepted in Bitcoin's mempool

BP2-020 - Slashing can be avoided with unbonding transactions

BP2-021 - Lack of transaction retry mechanism may result in delayed transactions

BP2-022 - Single submitter can submit all checkpoints due to delayed sealed checkpoint processing

BP2-023 - Submitter transactions may not be relayed due to very low change output value

BP2-024 - Well-positioned attacker can obtain critical signatures

BP2-025 - Finality provider can miss signature due to overflow

BP2-026 - Stake can get stuck with transactions bigger than 8 kilobytes

BP2-027 - Stake can get stuck due to race condition when delegating to a slashed finality provider

BP2-028 - Change in parameters leads to slashing being missed

BP2-029 - Attacker can avoid slashing in BTC by using wrapped messages

BP2-030 - Attacker can post fake stake in Babylon by forking Bitcoin

BP2-031 - Chain bloated by checkpoints that are never forgotten

BP2-032 - Users may have their transactions rejected due to the generation of dust outputs

BP2-033 - Unchecked user input allows retrieving wrong global parameters

BP2-034 - Fixed request ID in Keystone wallet requests prevents request-response association

BP2-035 - Malicious external API can crash staking service

BP2-036 - Inability to replay a single unprocessable message can prevent API server from starting

BP2-037 - Generated protobuf code cannot be checked for integrity

BP2-038 - Bitcoin transactions might be rejected due to insufficient minimum feerate

BP2-039 - Library users can create invalid staking transactions

BP2-040 - Attackers can eclipse node and post a fake Bitcoin chain

BP2-041 - Malicious indexer instance can crash vigilante staking tracker

BP2-042 - BLS private key exposed at rest

BP2-043 - Lack of differential testing between implementations of Bitcoin script libraries

BP2-044 - Proof-of-Possession (PoP) allows for reuse

6. Disclaimer

1. Executive Summary

In **December 2024**, **Babylon Labs** engaged Coinspect to perform a Source Code Audit of the Babylon project and the changes made for the Phase-2 of the chain.

Babylon's Phase 2 enables Bitcoin stake to secure the Babylon Genesis chain. Stakers and finality providers need to opt-in into securing the Babylon Genesis network. It provides features for stakers of Phase 1 to migrate their stake to Phase 2.

✓ Solved	⚠ Caution Advised	✗ Resolution Pending
High 4	High 0	High 0
Medium 13	Medium 0	Medium 0
Low 7	Low 0	Low 0
No Risk 20	No Risk 0	No Risk 0
Total **44**	Total **0**	Total **0**

The review resulted in 4 high severity issues: `BP2-001` details how a minority of stakers can cause inconsistent finalization. `BP2-003` describes how an attacker can avoid checkpointing changes to the validator set. `BP2-017` shows that the nodes on the network will crash soon after slashing a validator. `BP2-029` shows how an adversarial finality provider can consistently avoid slashings in BTC. All of these risks have been mitigated.

2. Summary of Findings

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

The total risk is determined by evaluating its impact and likelihood within the context of the system’s threat model:

- **Impact** (Low to High): Measures the potential consequences of a successful attack on the system’s confidentiality, integrity, and availability.
- **Likelihood** (Low to High): Estimates the probability of an attack’s success, factoring in technical complexity, exploitability, and cost-benefit considerations for an attacker.

Combining impact and likelihood produces the overall risk level: higher impact and likelihood lead to greater risk, while lower values reduce it. This approach prioritizes vulnerabilities consistently within a single security report.

2.3 Solved issues & recommendations

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

Id	Title	Risk
BP2-001	Minority of stakers can cause inconsistent finalization	High
BP2-003	Attacker can avoid checkpointing state by sending more than one message	High
BP2-017	Slashing a validator will cause the node to crash	High
BP2-029	Attacker can avoid slashing in BTC by using wrapped messages	High
BP2-004	Attacker can spam mempool with Bitcoin related messages	Medium
BP2-005	Attacker can avoid checkpointing state by using WASM contracts	Medium

BP2-009	Attacker can avoid checkpointing with embedded messages in WASM contracts	Medium
BP2-010	Attacker can stall chain with WASM contracts	Medium
BP2-011	Governance can approve messages that bypass checkpointing	Medium
BP2-012	Submitter will crash due to unhandled null return value	Medium
BP2-018	Wrong calculation of default transaction fee bump prevents bumping transactions	Medium
BP2-019	Replacement transactions might not be accepted in Bitcoin's mempool	Medium
BP2-020	Slashing can be avoided with unbonding transactions	Medium
BP2-024	Well-positioned attacker can obtain critical signatures	Medium
BP2-026	Stake can get stuck with transactions bigger than 8 kilobytes	Medium
BP2-028	Change in parameters leads to slashing being missed	Medium
BP2-030	Attacker can post fake stake in Babylon by forking Bitcoin	Medium
BP2-006	Default transaction fee cap may delay Bitcoin transaction confirmation	Low
BP2-007	User can waste time and funds due to generation of non-standard transactions	Low
BP2-013	Adversary can force Babylon node to waste resources by reprocessing known blocks	Low
BP2-014	Vigilante Reporter will eventually crash due to unbounded map growth	Low
BP2-031	Chain bloated by checkpoints that are never forgotten	Low
BP2-032	Users may have their transactions rejected due to the generation of dust outputs	Low
BP2-038	Bitcoin transactions might be rejected due to insufficient minimum feerate	Low

BP2-002	Recursion in AnteHandler allows for potentially arbitrary stack-depth	None
BP2-008	Unsupported TLS connection to bitcoind nodes	None
BP2-015	Transaction execution at risk due to decreased feerate	None
BP2-016	Bitcoin transactions not fully validated for standardization in Babylon node	None
BP2-021	Lack of transaction retry mechanism may result in delayed transactions	None
BP2-022	Single submitter can submit all checkpoints due to delayed sealed checkpoint processing	None
BP2-023	Submitter transactions may not be relayed due to very low change output value	None
BP2-025	Finality provider can miss signature due to overflow	None
BP2-027	Stake can get stuck due to race condition when delegating to a slashed finality provider	None
BP2-033	Unchecked user input allows retrieving wrong global parameters	None
BP2-034	Fixed request ID in Keystone wallet requests prevents request-response association	None
BP2-035	Malicious external API can crash staking service	None
BP2-036	Inability to replay a single unprocessable message can prevent API server from starting	None
BP2-037	Generated protobuf code cannot be checked for integrity	None
BP2-039	Library users can create invalid staking transactions	None
BP2-040	Attackers can eclipse node and post a fake Bitcoin chain	None
BP2-041	Malicious indexer instance can crash vigilante staking tracker	None
BP2-042	BLS private key exposed at rest	None

BP2-043	Lack of differential testing between implementations of Bitcoin script libraries	None
BP2-044	Proof-of-Possession (PoP) allows for reuse	None

3. Scope

The review lasted for 8 weeks, with a start date of December 23, 2024. At the start of the review, the scope was set to the following Babylon core repositories.

1. `https://github.com/babylonlabs-io/babylon/` at commit `a3b749d7cd6f9d46fc484508bba9aea719eb94b2`.
2. `https://github.com/babylonlabs-io/vigilante/` at commit `5d02378e4fdef27826b5d5ee6bf053d1516f8cfb`.
3. `https://github.com/babylonlabs-io/btc-staker/` at commit `b3c16973078c4b95e8adbebb8c83e0f404aec937`.
4. `https://github.com/babylonlabs-io/finality-provider/` at commit `f57fbddf218a84faa2c2ced868485de725364f3f`.
5. `https://github.com/babylonlabs-io/covenant-emulator/` at commit `817dbba627f828e265f790dcce08f77c94c4c358`.

On **Week 6** of the review, the scope of the web services repositories was set and their review started. The web services repositories were:

1. `https://github.com/babylonlabs-io/simple-staking` at commit `ddac0e8860d4ccf19462398cb0e8e60e8952ee26`.
2. `https://github.com/babylonlabs-io/bbn-core-ui` at commit `9a6a5548d36049884d32d48fc7cb55c88e2152f9`.
3. `https://github.com/babylonlabs-io/babylon-wallet-connect/` at commit `411e125f4fc386d3bda50c7c6ab470e6da439ee4`.
4. `https://github.com/babylonlabs-io/btc-staking-ts` at commit `de1d1cff3ef4ad45b5c61f0d7b11fd4479de278a`.
5. `https://github.com/babylonlabs-io/staking-api-service` at commit `eb99c0d24cd55fff474755cf886eed281333adcc`.
6. `https://github.com/babylonlabs-io/babylon-staking-indexer/` at commit `46d1de1efb181fcce4d29e70deac42b28ac9e9b8`.
7. `https://github.com/babylonlabs-io/staking-expiry-checker` at commit `c46282e02cf857b2f3ce69abf9132b3415097d39`.
8. `https://github.com/babylonlabs-io/staking-queue-client` at commit `c4b08ada1f40852bdeecee550866eae308b52616`.
9. `https://github.com/babylonlabs-io/babylon-proto-ts` at commit `11d093c77ae3b7da5594fabfa34108ab41c2e6f7`.

See the `4. Assessment` section for details on the difference between the two sets of repositories.

Towards the end of the engagement, the Babylon Labs team requested a review of the following Pull Requests:

1. `https://github.com/babylonlabs-io/babylon/pull/467`, which addresses a BLS keystore improvement. This was merged into the `main` branch at commit `d065cdd9d7f1219fece38cb678d1233566cf530b`.
2. `https://github.com/babylonlabs-io/vigilante/pull/211`, introducing an indexer service to the Bitcoin staking tracker module. These changes were merged into the `main` branch at commit `d1fbd204c307f27c240799df406a79f8ac86e03`.

After the engagement ended, the Babylon Labs team requested further review of two additional pull requests related to changes in the proof-of-possession verification. The pull requests were:

1. `https://github.com/babylonlabs-io/btc-staking-ts/pull/72/files`
2. `https://github.com/babylonlabs-io/babylon/pull/706`

All pull requests in scope were reviewed independently of the rest of the components and only the `diff` shown in the pull request itself was in scope.

3.1 Review of fixes

The review of fixes was made during and after the main review. Each fix was separated into a different PR by the Babylon Labs team. Coinspect analyzed each pull request to check if the changes mitigated the relevant issue.

The details of the PRs can be found on the `Status` section of each issue.

4. Assessment

The Babylon Genesis chain aims to act as a security and liquidity coordination layer onboarding bitcoin liquidity and security to the decentralized world. Phase-2 is the next step towards this vision by enabling bitcoin security and liquidity to be onboarded on the Babylon Genesis chain. This is achieved by leveraging several different strategies: Bitcoin staking scripts assert that stake will be locked for slashing if needed, Extractable-One-Time-Signatures are used to slash stakers even when they double-sign on a chain that does not natively have access to the staked funds, and Bitcoin timestamping is used to assert that a canonical chain is inscribed in Bitcoin and cannot be reversed without controlling a majority of Bitcoin Proof-of-Work.

The review encompasses periphery components besides the node itself, such as the `vigilante` project and the `covenant-emulator`. While these projects are not part of the node, they are security critical: the Babylon chain depends on the existence of at least one honest, well-behaved `vigilante` to listen for slashing events from the Babylon chain and leverage the information at its disposal to slash double-signers in Bitcoin. The `covenant-emulator` is run by an allowlisted set of entities and needs an honest majority: if most of the covenant set members are not honest, they can prevent new stakes or collude with finality providers so as to avoid slashings. The `finality-provider` component is also important, as it needs to behave correctly to avoid double-signing by mistake.

Additionally, The BTC staker software enables Bitcoin staking through a daemon (`stakerd`) that connects to Bitcoin and Babylon nodes, plus a CLI tool (`stakercli`) for user operations. The CLI lets users stake, withdraw, and unbond funds while monitoring finality providers in Babylon. It requires a Bitcoin node with `bitcoind` recommended over `btcd`. Coinspect mainly focused on examining limitations that could prevent users from executing transactions at desired times, as well as vulnerabilities in parameter validation that could result in unintended transaction outcomes.

The `vigilante`'s submitter module ensures the Babylon network periodically submits checkpoints to the Bitcoin network. Since each checkpoint exceeds Bitcoin's `OP_RETURN` 80-byte limit, it is split into two BTC transactions. The submitter repeatedly fetches `SEALED` checkpoints from Babylon and submits them to Bitcoin until they are successfully recorded on Bitcoin while minimizing duplicate submissions and transaction fees. Coinspect covered issues related to race conditions, the creation and submission of correct Bitcoin transactions, error handling, and denial-of-service risks.

On the other hand, the `vigilante`'s reporter module is responsible for forwarding Bitcoin headers and checkpoints to the Babylon network, keeping Babylon's

Bitcoin header chain updated. Upon detecting a new Bitcoin block, the reporter extracts the block header and any associated checkpoint, wraps them into transactions, and submits them to Babylon. The security review examined risks related to external disruptions in checkpoint transactions processing and reporting, as well as ensuring proper startup behavior, including synchronizing with an up-to-date Bitcoin node and accurately reporting all Bitcoin blocks to Babylon.

It is worth pointing out that it is unknown how, when, and how much rewards will be distributed to submitters and reporters, but rewards should account for BTC transaction fees, and the design must ensure that rewards exceed the costs incurred. Additionally, Coinspect found multiple `TODO` comments. Several issues were identified as having a root cause related to `TODO` notes. The total test coverage for this repository was `28.7%`. Coinspect recommends increasing the coverage of the unit tests to prevent problems being introduced in subsequent changes as this repository is one of the most critical for maintaining Babylon's security guarantees.

The `covenant-emulator` is another important part of the system. The `covenant-emulator` is a program that is intended to be run by a set of members in a committee. The main job of the committee is to prevent stakers from providing staking transactions that have already been `unbonded` before the system had time to process them. The committee will only provide signatures for the unbonding transactions once the intention to stake has been announced. **It is important to note that, in the presence of an evil majority of committee members, the security assumptions of the system no longer hold**. This evil committee would be able to collude and create finality providers that can avoid slashing when double signing. It is also important to note that if the committees stop responding to signing requests the whole system will stop processing new stake. Nevertheless, the Bitcoin stake of delegators that have given funds to an honest finality provider would be safe.

This review also included web components such as the `staking-api-service` and `simple-staking`. These repositories serve as interfaces and utilities that aim to provide a smooth experience for stakers and other users of the chain. While not critical compared to the chain-related components, these components need to behave correctly to prevent users from staking with incorrect data or being victim of phishing attempts. They also need to consider potential attackers that want to perform a denial of service.

Coinspect started analyzing these web components during the last two weeks of the review effort. These web components have a different threat model than the chain itself. The most important threats are:

1. Web vulnerabilities such as Cross-Site Scripting or Open Redirects, which allow attackers to trick users into singing malicious transactions
2. Denial-of-service attacks against Babylon applications.
3. Safety issues where a user can, by mistake, create and broadcast a Bitcoin transaction that will be then rejected by the Babylon chain.

4. Bugs in the interactions with the wallets that lead to mistakes in the creation of Bitcoin transactions.
5. Supply-chain issues that lead to a compromised frontend being presented to users.

As the wallets are out of scope for this particular review, the ability to detect problems in category (4) is limited. On the same note, potential supply-chain or infrastructure-related attacks are difficult to detect from the application's codebase. With this in mind, Coinspect has crafted some general advice to mitigate these risks (see `4.2: Mitigating supply chain and infrastructure risks`).

One of the most important repositories of the web application set is `btc-staking-ts`. This library builds Bitcoin transactions that stakers will use to stake and unbond, and that `vigilante`s can use to slash in case of equivocations. Note that the library is a mirror in Typescript of the Go library found at `babylon/btcstaking`. As such, Coinspect **strongly recommends** that differential testing be implemented between the two, as this is the most effective way to detect mismatches between the two implementations. This recommendation is reflected in `BP2-043`.

Another special repository is the `babylon-proto-ts`, which is generated from the `protobuf` files of the Babylon node. While Coinspect attempted to reproduce the generated code, the content of `babylon-proto-ts` did not match the results of regeneration. The risk of an exploit being hidden by malicious actors in the generated code is documented in the informational issue `BP2-037`.

Validator secret management

Currently, Babylon node operators can configure validators using the storage methods specified in the CometBFT documentation. One of the methods involves storing private keys in plaintext within a file on the validator server. However, this approach introduces a security risk: if an attacker gains access to a validator's host or their filesystem backup, they can extract the validator's private key. A similar issue was identified during the review of a BLS keystore-related pull request, as outlined in `BP2-042`.

Notably, CometBFT supports an alternative mechanism that leverages a remote signer. Publicly available implementations such as Tendermint KMS, are typically used alongside a Cloud HSM setup to mitigate the risk of key exposure at rest. However, even if a remote signer with HSM is implemented, certain Babylon node functions still require the plaintext private key to reside in a file, as they invoke the `LoadConsensusKey` function.

Coinspect recommends providing support for validators that want to decouple the BLS key management from the node by supporting a remote signer interface for BLS Keys, similar to what Cosmos supports for the validator key. Babylon's validator should be made aware of the intrinsic risks of handling a private key that is automatically available. Recommendations for validators should describe:

- Current CometBFT requirements and limitations.
- The necessity for two different keys (validator key, and BLS key).
- Resource requirements to implement and sustain the strategy, such as a Cloud HSM.
- The benefits of protecting the secrets at rest.

Babylon Labs has stated that including remote BLS key management is something that is planned in their roadmap.

Proof-of-Possession (PoP) review

Towards the end of the project, Babylon Labs requested a review of Pull Requests #706 and #72. These updates enable Proof-of-Possession (PoP) signing using BIP-322 for Taproot and native SegWit addresses. PoPs serve to verify control over a Bitcoin address by a Babylon actor and are required for creating finality providers and registering Bitcoin delegations. With these changes, the PoP is now expected to include the Babylon address. Coinspect's review also examined the code handling PoP verification (`CreateBTCDelegation` and `CreateFinalityProvider`) to ensure no unintended side effects.

4.1 Security assumptions

- The Bitcoin network is assumed to be reliable and it is expected its hashrate will not fluctuate drastically.
- The Bitcoin network is assumed to not be censoring or delaying inclusion of certain transactions.
- At least one vigilante program is running and it gets its information from non-adversarial Bitcoin and Babylon nodes.
- At least two thirds of the finality providers (when weighted by BTC stake delegated to them) are live to enable finalization of blocks.
- At least two thirds of the native CometBFT validators are live and produce new blocks.
- The Bitcoin and Babylon nodes used by the staker daemon are trusted and operating correctly.
- The majority of the covenant committee is live and honest.

4.2 Mitigating supply chain and infrastructure risks

A `dApp` is only as secure as its supply chain and the network it runs on. If an attacker can compromise infrastructure or code dependencies in such a way that users are delivered a compromised frontend when they access the domain of the `dApp`, the attacker has absolute control over the actions of the frontend and can perform a myriad of attacks.

These attacks are complex and cannot be detected on the application's codebase. As such, Coinspect is providing Babylon with a set of general guidelines to follow to mitigate these risks. Nevertheless, constant monitoring is needed.

- **Local-first**: an user should be able to download the repository from Github and run the frontend locally.
- **Prioritize reproducibility**: that same user should be able to easily compare between their version of the frontend and the one hosted by Babylon.
- **2FA is a must**: use two-factor authentication for all accounts that have access to the cloud environment, even if they cannot access the host where the application is running. Avoid using SMS as a second factor.
- **Implement constant monitoring**: to detect changes in DNS records, new scripts being loaded by the frontend, changes to the CDN data or new deployments on the relevant hosts.

5. Detailed Findings

BP2-001

Minority of stakers can cause inconsistent finalization

Status **Solved**	Risk **High**
✓	Impact **High**
Resolution **Fixed**	Likelihood **High**

Location

`babylon/x/finality/keeper/tallying.go`

Description

A set of evil finality providers that control 1/3 of the stake can prevent a block from being finalized and then finalize subsequent blocks. This breaks a core blockchain invariant: if a block at height `n` is finalized, then a block at height `n-1` should also be finalized.

Because of this issue, a group of finality providers in control of 1/3 of the stake can observe a Tendermint block at height h and avoid providing the *finality gadget* signature for it with their EOTS key. One the next block, they do provide their EOTS. This means that, from Babylon's point of view, block h+1 is finalized but h is not.

Being a core invariant, the problem has several consequences. Conceptually, it allows evil finality providers to vote for non-canonical blocks while maintaining the illusion that the chain is still being finalized, as the LastFinalizedHeight will be updated.

More practically, the rewards depend on this invariant. In babylon/x/finality/keeper/rewarding.go, the rewarding process will halt indefinitely as soon as it hits the first non-finalized block.

```
for height := nextHeightToReward; height <= uint64(targetHeight);
height++ {
    block, err := k.GetBlock(ctx, height)
    if err != nil {
        panic(err)
    }
    if !block.Finalized {
        break
    }
    k.rewardBTCStaking(ctx, height)
    nextHeightToReward = height + 1
}
```

The root cause is a wrongly-placed break inside a switch statement in babylon/x/finality/keeper/tallying.go:

```
switch {
    case fpSet != nil && !ib.Finalized:
        // has finality providers, non-finalised: tally and try to
finalise the block
        voterBTCPKs := k.GetVoters(ctx, ib.Height)
        if tally(fpSet, voterBTCPKs) {
            // if this block gets >2/3 votes, finalise it
            k.finalizeBlock(ctx, ib)
        } else {
            // if not, then this block and all subsequent blocks should
not be finalised
            // thus, we need to break here
            break
}
```

Recommendation

Make the iteration that finalizes blocks end as soon as the first block that does not reach a super-majority of voting power is encountered.

Status

Fixed in pull request #375. The loop now breaks immediately upon finding a non-finalized block.

BP2-002

Recursion in AnteHandler allows for potentially arbitrary stack-depth

Status

Solved

✓

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
babylon/x/epoching/keeper/drop_validator_msg_decorator.go
```

Description

The `NewDropValidatorMsgDecorator` antehandler uses recursion to check the `InternalMsgs` of the `authz.MsgExec` messages. This allows an attacker to send an `authz.MsgExec` message which contains an embedded `authz.MsgExec` and so on and so forth.

While no impact was observed with current parameters, if the network increased the maximum gas available on blocks or node operators limit go stack size this would allow an attacker to cause issues due to the stack growing without control, forcing the program to stop. Note that while the attacker would need funds to pass the gas checks (as part of the gas is charged by transaction size), the attacker would not spend the funds, as the node would hit the stack limit before the transaction is inserted in the mempool and thus would not be valid.

```
// check if any of the internal messages is a validator-related message
for _, internalMsg := range internalMsgs {
    // recursively validate the internal message
    if err := qmd.ValidateMsg(internalMsg); err != nil {
        return err
    }
}
```

Recommendation

Change the recursive logic to an iterative one. Consider putting a limit on the amount of nested messages supported by the `AnteHandler`.

Status

Fixed in pull request #468. The `DropValidatorMsgDecorator` does not exist anymore.

BP2-003

Attacker can avoid checkpointing state by sending more than one message

Status

Solved

✓

Resolution

Fixed

Risk

High

Impact

High

Likelihood

High

Location

`babylon/x/epoching/keeper/drop_validator_msg_decorator.go`

Description

An attacker can bypass the `DropValidatorMsgDecorator` antehandler by sending a transaction with more than one message. The `DropValidatorMsgDecorator` is supposed to prevent users from making stake-related changes to the blockchain state between epochs, as it is imperative for the protocol that all stake-related state is checkpointed into Bitcoin. By leveraging this attack, adversaries can change stake-related state and avoid it being checkpointed, as the message is executed immediately.

The root cause of the is a missing `err != nil` check in the iterator that goes through the messages of the transaction:

```
func (qmd DropValidatorMsgDecorator) AnteHandle(ctx sdk.Context, tx
sdk.Tx, simulate bool, next sdk.AnteHandler) (newCtx sdk.Context,j err
error) {
   // skip if at genesis block, as genesis state contains txs that
bootstrap the initial validator set
   if ctx.BlockHeight() == 0 {
       return next(ctx, tx, simulate)
   }
   // after genesis, if validator-related message, reject msg
   for i, msg := range tx.GetMsgs() {
       err := qmd.ValidateMsg(msg)
       return ctx, err
   }


   return next(ctx, tx, simulate)
}
```

Because the `AnteHandle` returns after the first message whether there is an error or not, an attacker can send a staking message as the second message in the transaction and have it executed immediately.

Recommendation

Add an `err != nil` check.

Status

Fixed in pull request #385.

Updated in pull request #468, which deleted the decorator itself.

BP2-004

Attacker can spam mempool with Bitcoin related messages

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

Medium

Likelihood

Medium

Location

```
babylon/x/epoching/keeper/drop_validator_msg_decorator.go
```

Description

An attacker can spam the mempool with invalid `MsgInsertBTCSpvProof` and `MsgInsertHeaders` messages as these messages will skip checks when being checked for mempool insertion.

The root cause of the issues is the code-section as `BP2-003`: because a return that is supposed to be called only in case of an error is invoked always in the `DropValidatorMsgDecorator`, it does not call the `next` callback; which should continue the check chain.

Because of this, the next ante handler is not called. In Babylon, the next ante handler configured after the `DropValidatorMsgDecorator` is the `BtcValidationDecorator`, which checks `MsgInsertBTCSpvProof` and `MsgInsertHeaders` messages during `CheckTx` and `ReCheckTx` operations.

Recommendation

Add an `err != nil` check.

Status

Fixed in pull request #385.

Updated in pull request #468, which deleted the decorator itself.

BP2-005

Attacker can avoid checkpointing state by using WASM contracts

Status

Solved

✓

Resolution

Fixed

Risk

Medium

▼
🟥🟥⬜

Impact

High

Likelihood

Low

Location

Description

An attacker can deploy a `cosmwasm` contract to interact directly with the Cosmos `staking` module, bypassing the `DropValidatorMsgDecorator` and the queuing mechanism implemented therein. The impact of this attack is exactly the same as `BP2-003`. The likelihood of this attack is considered `low` only because the deployment of `cosmwasm` contracts is whitelisted on Babylon mainnet. Nevertheless, the Babylon Labs team expressed interest in removing the whitelist, so Coinspect also researched potential mitigations and showed that using the `capabilities` features of `cosmwasm` would not serve as a fix.

The issue lies in how `cosmwasm` contracts interact with the underlying chain. `cosmwasm` uses a message system that defers calls from a contract to other modules until the end of the message execution. To execute the calls, `cosmwasm` simply dispatches them through the `MsgRouter` of the Cosmos application.

Because the `staking` module is a registered service of Babylon's application, the router will dispatch calls to it directly. The submessage will **not** go through any ante handler.

While removing the `staking` capability from the capabilities list on the `wasm` keeper appears as a possible mitigation, it is trivial to bypass for an attacker: `cosmwasm` checks for capabilities with a hardcoded string on the `wasm` binary but still exposes all functionality to the binary regardless of the capabilities it announces. If the `staking` capability were removed, an attacker would only have to edit the `requires_staking` string out of the `wasm` binary to be able to deploy and execute the code.

Recommendation

Maintain the whitelist to deploy contracts and review each contract to make sure it does not communicate with the native `staking` module.

Alternatively, consider removing the native `staking` module from the application's router. This way submessages sent by `cosmwasm` contracts will not be able to find the `staking` module. Nevertheless, this implies several changes to the Babylon codebase, as Babylon's app now relies on the existence of said routes for certain operations such as genesis creation.

See `BP2-009` for more possible mitigations.

Status

Fixed in pull request #468. The process by which services are registered in Cosmos has been modified so as to avoid registering the `staking` module in the router. This makes the `staking` routes inaccessible via the message server.

Running a `cosmwasm` contract that communicates with the staking module now results in a `can't route message` error:

```
Error: rpc error: code = Unknown desc = rpc error: code = Unknown desc
= failed to execute message; message index: 0: dispatch: submessages:
can't route message
delegator_address:"bbn14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmf
vw9sw76fy2"
validator_address:"bbnvaloper1a0r5s47deargeftzhjcgrmra8cfu2c5d6933ys"
amount:<denom:"ubbn" amount:"10000000" > : unknown request
[CosmWasm/wasmd@v0.53.0/x/wasm/keeper/handler_plugin.go:125] with gas
used: '144688': unknown request
```

BP2-006

Default transaction fee cap may delay Bitcoin transaction confirmation

Status

Solved

✓

Resolution

Fixed

Risk

Low

▼

🟥□□

Impact

Medium

Likelihood

Low

Location

```
btc-staker/staker/feeestimator.go
vigilante/submitter/relayer/relayer.go
```

Description

Transactions may be delayed when the default fee rate cap is used and it is insufficient for the current network fee, potentially causing stakers to miss staking deadlines or limits.

The staker CLI's fee estimator service enforces a default `MaxFeeRate` of 25 sat/vbyte, which is applied when the estimated network fee exceeds this limit.

```
if estimatedFee > e.MaxFeeRate {
   e.logger.WithFields(logrus.Fields{
       "maxFeeRate": e.MaxFeeRate,
       "estimated":  estimatedFee,
   }).Debug("Estimated fee is higher than max fee rate. Using max fee
rate")
```

```
   return e.MaxFeeRate
}
```

It is worth mentioning that users can override the `MaxFeeRate` by specifying a custom value in the configuration file, although this is optional.

This capped feerate is applied to staking transactions in the `StakeFunds` function:

```
feeRate := app.feeEstimator.EstimateFeePerKb()


app.logger.WithFields(logrus.Fields{
   "stakerAddress": stakerAddress,
   "stakingAmount": stakingInfo.StakingOutput,
   "fee":           feeRate,
}).Info("Created and signed staking transaction")


req := newOwnedStakingCommand(
   stakerAddress,
   stakingInfo.StakingOutput,
   feeRate,
...
```

A similar issue arises in the relayer code of the Vigilante Submitter module. The following snippet, taken from the `getFeeRate` function, determines the fee rate for Bitcoin checkpoint segment transactions. If this fee rate is capped below the current network fee rate, the Submitter may fail to submit the checkpoint in time:

```
cfg := rl.GetBTCConfig()
if feePerKVByte > cfg.TxFeeMax {
    rl.logger.Debugf("current tx fee rate is higher than the maximum tx
fee rate %v, using the max", cfg.TxFeeMax)
    feePerKVByte = cfg.TxFeeMax
}
if feePerKVByte < cfg.TxFeeMin {
    rl.logger.Debugf("current tx fee rate is lower than the minimum tx
fee rate %v, using the min", cfg.TxFeeMin)
    feePerKVByte = cfg.TxFeeMin
}
```

Recommendation

Communicate clearly to users the importance of setting their own caps and what the defaults currently are so they can create a fee strategy that satisfies their needs.

As a default is needed, set it to be more lax. The current 25 has been surpassed already in periods of high BTC activity. In general, a default of 200 sat/vbyte is reasonable, as it is high enough to serve even in periods of very high activity while protecting the operator from unreasonable losses in fees. Nevertheless, it is recommended that Babylon perform their own calculation to establish a maximum amount for users to use in fees by default and their average transaction virtual size.

Status

Fixed in#154 for `btc-staker` and #257 por `vigilante`. The new default is of 200 sat/vbyte.

BP2-007

User can waste time and funds due to generation of non-standard transactions

Status

Solved

✓

Resolution

Fixed

Risk

Low

Impact

Low

Likelihood

Low

Location

```
btc-staker/walletcontroller/transaction.go
btc-staker/staker/types.go
```

Description

A user can waste time and, as a result, get worse fees for their BTC transaction because the BTC staker code allows for the creation of non-standard transactions. Even though non-standard transactions are still valid, they will not be accepted by most miners. As a result, an user that wants to stake at a certain time will get delayed until they can diagnose and fix the issue, potentially resulting in worse fees if the network fees are increasing.

Coinspect noticed that the `buildTxFromOutputs` function in charge of creating transactions lacks checks regarding the change amount exceeding the minimum dust value, as well as transaction size checks. Additionally, no checks were observed on this function's return value down the execution path.

A similar situation occurs with the `createUndelegationData` and `createSpendStakeTx` functions from the `types.go` file.

Recommendation

Verify that transactions generated by the staker CLI meet the standard transaction policy. Specifically, add checks to prevent the component from creating outputs with dust values, as well as the transactions' size falling within the expected limits.

Status

Fixed in PR #146. Now the staker verifies that the generated transactions do satisfy the standard requirements, including slashing transactions.

BP2-008

Unsupported TLS connection to bitcoind nodes

Status

Solved

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
btc-staker/staker/feeestimator.go
vigilante/btcclient/client_wallet.go
```

Description

TLS connections to `bitcoind` nodes are not supported. While `bitcoind` itself does not natively support TLS, it can be configured behind a TLS proxy.

As illustrated below, the default RPC connection settings explicitly disable TLS connections to `bitcoind` nodes, and this restriction cannot be overridden through user configuration:

```
case types.BitcoindNodeBackend:
    rpcConfig := rpcclient.ConnConfig{
        Host:                 cfg.Bitcoind.RPCHost,
        User:                 cfg.Bitcoind.RPCUser,
        Pass:                 cfg.Bitcoind.RPCPass,
        DisableConnectOnNew:  true,
        DisableAutoReconnect: false,
        DisableTLS:           true,
        HTTPPostMode:         true,
    }
```

A similar situation occurs in the `NewWallet` from the Vigilante repository as shown below.

```
connCfg := &rpcclient.ConnConfig{
    // this will work with node loaded with multiple wallets
    Host:         cfg.BTC.Endpoint + "/wallet/" + cfg.BTC.WalletName,
    HTTPPostMode: true,
    User:         cfg.BTC.Username,
    Pass:         cfg.BTC.Password,
    DisableTLS:   true,
}
```

Recommendation

Introduce an option to allow users to enable TLS connections to `bitcoind` nodes.

Status

Acknowledged. Babylon Labs team is tracking work to tackle these issues in the future in #152 and #254.

BP2-009

Attacker can avoid checkpointing with embedded messages in WASM contracts

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

Description

An attacker can send messages directly to the `staking` module and avoid the `epoching` queue by sending an `authz.MsgExec` that contains a delegated call to the `staking` module.

This issue is very similar to `BP2-005`. Nevertheless, the vector is slightly different. Instead of sending a `staking` message directly, the attacker would wrap it in a `auth.MsgExec` or through the `gov/` module.

The difference is relevant as, depending on the mitigation used for `BP2-005`, this issue might still be exploitable even if `BP2-005` is fixed.

Recommendation

The most direct and pervasive mitigation for the family of issues relating to communicating with the staking module without waiting for the epoch to finish is to set the base application router to not contain the routes to the staking module, as the authz module interacts with other modules via the baseapp.Router and dispatches the messages through it, see cosmos-sdk/x/authz/keeper/keeper.go:

```
handler := k.router.Handler(msg)
if handler == nil {
   return nil, sdkerrors.ErrUnknownRequest.Wrapf("unrecognized message
route: %s", sdk.MsgTypeURL(msg))
}


msgResp, err := handler(sdkCtx, msg)
if err != nil {
   return nil, errorsmod.Wrapf(err, "failed to execute message; message
%v", msg)
}
```

The gov modules does the same, see cosmos-sdk/x/gov/abci.go

```
// execute all messages
for idx, msg = range messages {
   handler := keeper.Router().Handler(msg)
   var res *sdk.Result
   res, err = safeExecuteHandler(cacheCtx, msg, handler)
   if err != nil {
       break
   }
   events = append(events, res.GetEvents()...)
}
```

Status

Fixed in pull request #468. See BP2-005.

BP2-010

Attacker can stall chain with WASM contracts

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Low

Location

Description

An attacker can stall the chain by creating a WASM contract that creates an excessively-nested recursive message. This is described in a public Cosmos advisory and Coinspect was able to partially reproduce the issue in a proof of concept. With Coinspect's proof of concept, the victim node stalls and is not able to produce a block in time, stalling the chain.

This issue is similar to the previously reported `BP2-002`, but by leveraging the fact that `cosmwasm` contracts bypass the transaction check antehandler, the nodes are impacted.

While in Coinspect tests the node eventually recovers and is able to continue producing and receiving blocks, the attacker can send another transaction executing the expensive process. It is possible that in different configurations the attack leads to a crash, as it depends on the memory available in the validator's host.

Recommendation

Upgrade Cosmos to `v0.50.11` or higher.

Status

Fixed in pull request #405. Cosmos version was upgraded to `v0.50.11`.

Proof of concept

A proof of concept was shared privately with the Babylon Labs team.

BP2-011

Governance can approve messages that bypass checkpointing

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Low

Location

```
x/epoching/keeper/drop_validator_msg_decorator.go
```

Description

An attacker can convince the governance to approve a malicious message that modifies stake-related state and bypasses the logic needed for it to be checkpointed.

The attack has the same impact as the other issues related to bypassing the `DropValidatorMsgDecorator`. In this scenario, the vector is different: while the decorator analyzes the `authz.MsgExec` recursively to catch an embedded message, it does not check for the `Messages` of the `MsgSubmitProposal`:

```go
// MsgSubmitProposal defines an sdk.Msg type that supports submitting
arbitrary
// proposal Content.
type MsgSubmitProposal struct {
    // messages are the arbitrary messages to be executed if proposal
```

```
passes.
   Messages []*types.Any `protobuf:"bytes,1,rep,name=messages,proto3"
json:"messages,omitempty"`
   ...
```

Each message is then executed if the proposal is approved in `cosmos-sdk/x/gov/abci.go`:

```
// execute all messages
for idx, msg = range messages {
    handler := keeper.Router().Handler(msg)
    var res *sdk.Result
    res, err = safeExecuteHandler(cacheCtx, msg, handler)
    if err != nil {
        break
    }


    events = append(events, res.GetEvents()...)
}
```

The likelihood of this issue is low as it requires the governance to be either complicit or duped into approving a malicious proposal. Nevertheless, this poses a real risk and other projects have in fact been the target of malicious proposals. See the Tornado Cash malicious proposal hack.

Recommendation

Check for `gov` messages in the `AnteHandler`. Take note that governance should get the same treatment as messages related to the `authz` module, including precautions detailed in `BP2-002`.

Status

Fixed in pull request #468. See `BP2-005`.

BP2-012

Submitter will crash due to unhandled null return value

Status
Solved

✓

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Medium

Location

```
vigilante/submitter/relayer/relayer.go
```

Description

The submitter will panic when attempting to increase the fee for a checkpoint transaction that has already been confirmed by the Bitcoin network. This is possible due to a race condition between the Bitcoin network itself and the vigilante process.

When the submitter sends the second segment of a Babylon checkpoint as a Bitcoin transaction, it is added to the mempool. Subsequently, the submitter monitors the transaction's status using the relayer's `MaybeResubmitSecondCheckpointTx` function. If the transaction remains unconfirmed by the Bitcoin network, this function attempts to increase the transaction fee to expedite its inclusion in a block.

However, if the function attempts to bump the fee for a transaction that is already confirmed, it results in a `panic`. This occurs because the resendSecondTxOfCheckpointToBTC function returns nil, nil when the transaction is confirmed (as shown in the second if in the snippet below).

```
func (rl *Relayer) resendSecondTxOfCheckpointToBTC(tx2
*types.BtcTxInfo, bumpedFee btcutil.Amount) (*types.BtcTxInfo, error) {
   _, status, err := rl.TxDetails(rl.lastSubmittedCheckpoint.Tx2.TxID,

rl.lastSubmittedCheckpoint.Tx2.Tx.TxOut[changePosition].PkScript)
   if err != nil {
       return nil, err
   }


   // No need to resend, transaction already confirmed
   if status == btcclient.TxInChain {
       rl.logger.Debugf("Transaction %v is already confirmed",
rl.lastSubmittedCheckpoint.Tx2.TxID)


       return nil, nil
   }
```

The panic occurs because this nil, nil return value is unsafely accessed later in the function, as shown below:

```
resubmittedTx2, err :=
rl.resendSecondTxOfCheckpointToBTC(rl.lastSubmittedCheckpoint.Tx2,
bumpedFee)
...
resubmittedTx2.TxID.String(),
strconv.Itoa(int(resubmittedTx2.Fee)),
```

This issue happens in a race condition scenario where the Bitcoin checkpoint transaction is not confirmed immediately, leaving the Babylon checkpoint in a SEALED status. If the Bitcoin transaction is eventually confirmed but the Babylon checkpoint status remains unchanged, the submitter will encounter a panic when attempting to bump the fee.

Recommendation

The MaybeResubmitSecondCheckpointTx function should immediately terminate if the transaction is already confirmed, as there is no need to bump the fee.

Also, consider improving the relayer module's unit testing suite.

Status

FIxed in pull request #154. The `MaybeResubmitSecondCheckpointTx` function now terminates if the second return parameter from `resendSecondTxOfCheckpointToBTC` (now called `maybeResendSecondTxOfCheckpointToBTC`) is `nil`.

BP2-013

Adversary can force Babylon node to waste resources by reprocessing known blocks

Status

Solved

✓

Resolution

Fixed

Risk

Low

▼

■□□

Impact

Low

Likelihood

Low

Location

`babylon/x/btclightclient/types/btc_light_client.go`

Description

An attacker can exploit the Babylon node to waste computational resources by reprocessing known Bitcoin blocks. The attacker is not charged gas for the operation. This vulnerability arises from an issue in the Bitcoin light client module, which incorrectly treats a given chain of headers as a fork, without recognizing that the chain is actually the canonical one. As shown in the code snippet below, an attacker can target the `else` branch by providing a chain of headers starting from any block previous to the current chain's tip.

```
firstHeaderOfExtensionChain := headers[0]
store := newStoreWithExtensionChain(readStore, headersLen)

if firstHeaderOfExtensionChain.PrevBlock.IsEqual(&currentTipHash) {
   // most common case - extending the current tip
   if err := l.processNewHeadersChain(store, currentTip, headers); err
```

```
!= nil {
       return nil, err
   }


   return &InsertResult{
       HeadersToInsert: toBTCHeaderInfos(store.headers),
       RollbackInfo:    nil,
   }, nil
} else {
   // possible new fork
   parentHash :=
bbn.NewBTCHeaderHashBytesFromChainhash(&firstHeaderOfExtensionChain.Pre
vBlock)
   forkParent, err := readStore.GetHeaderByHash(&parentHash)
```

It is important to note that to trigger this vulnerability, a valid Bitcoin block must be submitted to ensure the chain has more work than the previous one.

The amount of work done by the Babylon nodes is only limited by the amount of blocks known by the chain and the Cosmos enforced maximum transaction bytes.

Only participants in the allow-list, which is configurable via parameters, can submit blocks. However, if this functionality is publicly accessible, anyone could trigger the vulnerability, significantly increasing the risk.

Furthermore, if the allow-list was removed and `cosmwasm` contracts could be used to interact with the light client module, an attacker could create a contract that performs the attack. This would increase the maximum number of headers that can be sent, which is currently capped by the Cosmos-enforced maximum transaction bytes limit. An increased number of headers would lead to significantly more wasted resources.

Recommendation

Before processing a potential new best fork, verify that the header at `firstHeaderOfExtensionChain` is not part of the chain containing the current tip. This ensures that the chain is properly validated as a new fork before any further processing.

Consider limiting the number of headers in a `MsgInsertHeaders` message.

Status

Fixed in pull request #446. The chain will reject forks that start with an already known header. The implementation uses the fact that a known and well-formed header does not return an error.

Proof of Concept

Coinspect prepared the following test, which inserts a block chain starting at this first block known by the node. It is recommended to follow the in-line comments.

```
func TestReprocessEntireChain(t *testing.T) {
   senderPrivKey := secp256k1.GenPrivKey()
   address, err :=
sdk.AccAddressFromHexUnsafe(senderPrivKey.PubKey().Address().String())
   require.NoError(t, err)
   r := rand.New(rand.NewSource(0x42))
   srv, blcKeeper, sdkCtx := setupMsgServer(t)
   ctx := sdk.UnwrapSDKContext(sdkCtx)
   _, chain := datagen.GenRandBtcChainInsertingInKeeper(
       t,
       r,
       blcKeeper,
       ctx,
       0,    // ! chain start height
       1000, // ! chain length
   )
   initTip := chain.GetTipInfo()


   checkTip(
       t,
       ctx,
       blcKeeper,
       *initTip.Work,
       initTip.Height,
       initTip.Header.ToBlockHeader(),
   )


   chainInfo := chain.GetChainInfo()
   firstKnownBlock := chainInfo[0]

   chainExtensionLength := uint32(1001) // MALICIOUS FORK LENGTH
   chainExtension := datagen.GenRandomValidChainStartingFrom(
       r,
       firstKnownBlock.Header.ToBlockHeader(), // OUR FORK STARTS AT
THE FIRST BLOCK KNOWN BY THE NODE
       nil,
       chainExtensionLength,
   )


   msg := &types.MsgInsertHeaders{Signer: address.String(),
       Headers: keepertest.NewBTCHeaderBytesList(chainExtension),
   }

   _, err = srv.InsertHeaders(sdkCtx, msg)
```

```
    require.NoError(t, err)
}
```

To execute this test, place it in `babylon/x/btclightclient/keeper/msg_server_test.go` and execute it by running `go test -v ./x/btclightclient/keeper -run TestReprocessEntireChain`.

BP2-014

Vigilante Reporter will eventually crash due to unbounded map growth

Status

Solved

Resolution

Fixed

Risk

Low

Impact

Low

Likelihood

Low

Location

`vigilante/types/ckpt_cache.go`

Description

The Babylon Vigilante Reporter will crash as the map that stores unmatched checkpoint segments is not periodically cleaned up. When unmatched segments accumulate, the resulting unbounded growth of data in these structures will lead to memory exhaustion on the host running the vigilante, ultimately causing it to crash. It is important to note that the `Match` function does remove segment pairs that successfully form a checkpoint:

```
// Remove the two ckptSeg in segMap
delete(c.Segments[uint8(0)], hash1)
delete(c.Segments[uint8(1)], hash2)
```

However, Coinspect did not identify logic to periodically clean up segments that cannot be matched to form a checkpoint.

As these single segments are obtained from Bitcoin transactions, it is unlikely for this attack to be achieved. This is due to the fact that an adversary would need to incur Bitcoin network fees to cause these maps to grow indefinitely. The likelihood is also lowered by the fact that the cache is re-initializated when the reporter detects it is out-of-sync with the Bitcoin node.

Recommendation

Implement a mechanism to clean up unmatched segments after a safe time period or a defined number of blocks have passed.

Status

Fixed in pull request #233. The checkpoints now have a Time-To-Live and are discarded once they expire.

BP2-015

Transaction execution at risk due to decreased feerate

Status

Solved

✓

Resolution

Fixed

Risk

None

▼

□□□

Impact

Recommendation

Likelihood

-

Location

`vigilante/submitter/relayer/relayer.go`

Description

The submitter's relayer generates transactions with a lower fee rate than intended due to the addition of a change output.

This happens because the relayer forces the inclusion of a change output for the first checkpoint segment transaction, even if one is not needed. This output is included after funding the transaction (selecting enough inputs to cover the value plus fees).

```
// 1. INPUTS ARE SELECTED
rawTxResult, err := rl.BTCWallet.FundRawTransaction(tx,
btcjson.FundRawTransactionOpts{
    FeeRate:        &feeRate,
    ChangePosition: &changePosition,
}, nil)
```

```
if err != nil {
   return nil, err
}


// 2. ADD CHANGE OUTPUT IF NOT PRESENT
// Ensure the firstTx has a change output, but we can skip this for the
second transaction
hasChange := len(rawTxResult.Transaction.TxOut) > changePosition
// Manually add a change output with 546 satoshis if needed
if !isSecondTx && !hasChange {
   changeAddr, err := rl.BTCWallet.GetRawChangeAddress(rl.walletName)
   if err != nil {
       return nil, fmt.Errorf("error getting raw change address: %w",
err)
   }


   changePkScript, err := txscript.PayToAddrScript(changeAddr)
   if err != nil {
       return nil, fmt.Errorf("failed to create script for change
address: %s, error: %w", changeAddr, err)
   }


   changeOutput := wire.NewTxOut(int64(dustThreshold), changePkScript)
   rawTxResult.Transaction.AddTxOut(changeOutput)
}
```

Although the change output added is slightly higher than a dust value, this amount is subtracted from the intended transaction fees, reducing the miner's incentive to include the transaction in a block. Additionally, the code does not account for the cost of adding this extra output.

This issue is considered informational because it's uncommon for a typical Bitcoin transaction to have no change output (referred to as an "exact payment" transaction).

Finally, it is important to note that once the change output is added, the hasChange variable which is later referenced in the function, is not updated. This causes the newly added change output to skip the different change output related validations.

Recommendation

Re-select transaction inputs after adding the change output.

Consider enhancing the unit testing suite for the submitter's relayer code to better handle such edge cases.

Status

Fixed in pull request #226. The inputs are reselected after constructing the change.

BP2-016

Bitcoin transactions not fully validated for standardization in Babylon node

Status

Solved

✓

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

`babylon/btcstaking/staking.go`

Description

The Babylon node lacks checks to fully assert that transactions submitted through `CreateBTCDelegation` are standard. The missing checks are not currently exploitable, although it is recommended to explicitly prevent the creation of transactions that do not conform to Bitcoin standard rules. This is particularly important for the slashing transaction, as the protocol's security depends on the ability of vigilantes to send slashing transactions to the Bitcoin network.

The missing validations are:

1. Transactions are not checked for standard witness data. A staker is free to put any data they want on the `TxIn[0].Witness` field of their transactions.

2. Transactions are not checked for minimum size. A staker could in theory send a transaction that is too small for it to be considered standard.

Both attacks have no impact in the current Babylon codebase. The standard-witness attack is thwarted by the fact that the witness data is *replaced*, not appended to, in the `vigilante`. The `vigilante` uses `BuildSlashingTxWithWitness` from the `btcstaking` library, which does:

```
slashingMsgTxWithWitness, err := tx.ToMsgTx()
if err != nil {
   return nil, err
}
slashingMsgTxWithWitness.TxIn[0].Witness = witness
```

Manipulating the transaction size is also currently impossible, due to the way the Babylon node requires the slashing transaction output scripts to match the expected ones, even for the *change output* of the slashing transaction. The added size of the two required scripts makes the transaction bigger than 64 bytes.

```
   if !bytes.Equal(slashingTx.TxOut[1].PkScript, si.PkScript) {
       return fmt.Errorf("invalid slashing tx change output pkscript, expected: %s, got: %s", hex.EncodeToString(si.PkScript), hex.EncodeToString(slashingTx.TxOut[1].PkScript))
   }
```

Recommendation

While the two missing validations are not exploitable as the current codebase stands, it is recommended to make them explicit and part of consensus.

Status

Acknowledged.

BP2-017

Slashing a validator will cause the node to crash

Status

Solved

Resolution

Fixed

Risk

High

Impact

High

Likelihood

High

Location

`babylon/x/incentive/keeper/btc_staking_gauge.go`

Description

Slashing a validator at height `h` will cause the node to crash at height `h + FinalitySigTimeout`. The node will process the slashing event immediately at height `h` causing the `FinalityProviderCurrentRewards` to be deleted. But the system assumes that `FinalityProviderCurrentRewards` exists when calling `RewardBTCStaking` at height `h + FinalitySigTimeout`.

To understand the issue, the interactions between the `x/incentive` module and the `x/finality` module need to be understood. One key data structure in this interaction is the `VotingPowerDistCache` which contains information about the finality providers at each height.

The `VotingPowerDistCache` is written via a `BeingBlocker` function. It depends on the events emitted by power-distribution-changing actions in other modules. One of these power distribution changes is slashing a finality provider, which calls the `IncentiveKeeper::FpSlashed` method *besides* modifying the power distribution cache for the `height` at which the slashing happened.

```
case *types.EventPowerDistUpdate_SlashedFp:
    // record slashed fps
    types.EmitSlashedFPEvent(sdkCtx, typedEvent.SlashedFp.Pk)
    fpBTCPKHex := typedEvent.SlashedFp.Pk.MarshalHex()
    slashedFPs[fpBTCPKHex] = struct{}{}
    fp := k.loadFP(ctx, fpByBtcPkHex, fpBTCPKHex)
    if err := k.IncentiveKeeper.FpSlashed(ctx, fp.Address()); err !=
nil {
            panic(err)
    }
```

The issue is that the `FpSlashed` method has a side-effect: it will immediately delete the slashed provider reward data:

```
// delete all reward tracker that correlates with the slashed finality
provider.
k.deleteKeysFromBTCDelegationRewardsTracker(ctx, fp,
keysBtcDelRwdTracker)
k.deleteAllFromFinalityProviderRwd(ctx, fp)
```

Consider then that `HandleRewarding` of the `x/finality` module is called only after all validators have had time to sign the blocks. That is: `HandleRewarding` is called not on the tip of the chain, but with a delay of `FinalitySigTimeout` blocks.

`HandleRewarding` ends up calling `RewardBTCStaking` with the `VotingPowerDistCache` of the block. It assumes that the keys from the finality provider still exist:

```
// btc_staking_gauge.go
if err := k.AddFinalityProviderRewardsForBtcDelegations(ctx,
fp.GetAddress(), coinsForBTCDels); err != nil {
    panic(err)
}

// reward_tracker.go
func (k Keeper) AddFinalityProviderRewardsForBtcDelegations(ctx
context.Context, fp sdk.AccAddress, rwd sdk.Coins) error {
        fpCurrentRwd, err := k.GetFinalityProviderCurrentRewards(ctx,
fp)
        if err != nil {
                return err
        }
```

But `GetFinalityProviderCurrentRewards` will not exist, because while processing the slashed event the keys were removed immediately. This causes `AddFinalityProviderRewardsForBtcDelegations` to return an error, which will cause `RewardBTCStaking` to `panic`.

Because `RewardBTCStaking` runs as part of an `EndBlocker` process, this `panic` is non-recoverable, and will cause a crash in the nodes that are processing this block.

Recommendation

Avoid modifying the storage immediately upon a slashing event. Instead, consider using the `IsSlashed` information on a finality provider to modify the storage only when the rewards for a slashed validator should be processed.

Status

Fixed in PR #372.

The Babylon Labs team found this issue independently and fixed it soon after this review started. Their mitigation strategy was to use the `IsSlashed` data to avoid adding the slashed finality providers to the `VotingPowerDistCache`.

BP2-018

Wrong calculation of default transaction fee bump prevents bumping transactions

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

Medium

Likelihood

Medium

Location

`vigilante/submitter/relayer/relayer.go`

Description

The relayer's Replace-by-Fee (RBF) mechanism for Bitcoin transactions will not work under the following scenarios:

1. When using the default `ResubmitFeeMultiplier` value.
2. When the current fee requirements exceed the fee chosen for the Replace-by-Fee transaction.

The Vigilante Submitter's attempts to ensure the inclusion of checkpoint transactions by Replacing-by-Fee (RBF) the second segment transaction. However, a wrong calculation of the new fee will prevent this mechanism from succeeding if using the default configuration value.

As shown in the following snippet, the `MaybeResubmitSecondCheckpointTx` function computes the new fee and later evaluates whether the bumped fee is enough for RBF.

```
bumpedFee := rl.calculateBumpedFee(rl.lastSubmittedCheckpoint)

// make sure the bumped fee is effective
if !rl.shouldResendCheckpoint(rl.lastSubmittedCheckpoint, bumpedFee) {
    return nil
}
```

However, the `calculateBumpedFee` function only multiplies the fee of the transaction to be replaced by a constant multiplier (`ResubmitFeeMultiplier`) passed from the configuration, without considering the current network fee conditions. By default, this constant is set to `1` (`DefaultResubmitFeeMultiplier`), meaning that the `calculateBumpedFee` function returns the fee of the transaction to be replaced.

```
func (rl *Relayer) calculateBumpedFee(ckptInfo *types.CheckpointInfo)
btcutil.Amount {
        return ckptInfo.Tx2.Fee.MulF64(rl.config.ResubmitFeeMultiplier)
}
```

Then, when the `shouldResendCheckpoint` is called, it will always return false by default, as `requiredBumpingFee` will be greater than `bumpedFee`.

```
// shouldResendCheckpoint checks whether the bumpedFee is effective for
replacement
func (rl *Relayer) shouldResendCheckpoint(ckptInfo
*types.CheckpointInfo, bumpedFee btcutil.Amount) bool { //ok
        // if the bumped fee is less than the fee of the previous
second tx plus the minimum required bumping fee
        // then the bumping would not be effective
        requiredBumpingFee := ckptInfo.Tx2.Fee +
rl.calcMinRelayFee(ckptInfo.Tx2.Size)

        rl.logger.Debugf("the bumped fee: %v Satoshis, the required
fee: %v Satoshis",
                bumpedFee, requiredBumpingFee)

        return bumpedFee >= requiredBumpingFee
}
```

Note however that the main problem is not the usage of a bad default fee multiplier but the lack of consideration of the current network fees for the Replace-by-Fee transaction. The result of the `calculateBumpedFee` function might not be enough to cover the Replace-by-Fee transaction fee under the current network conditions.

Recommendation

Compute the replacement transaction fee considering the current network fee. This resulting value can be also multiplied by a fixed number to ensure its prompt inclusion.

Status

Fixed in pull request #223. The `calculateBumpedFee` now considers the current `feeRate`.

BP2-019

Replacement transactions might not be accepted in Bitcoin's mempool

Status

Solved

✓

Resolution

Fixed

Risk

Medium

🟥🟥⬜

Impact

Medium

Likelihood

Medium

Location

```
vigilante/submitter/relayer/relayer.go
```

Description

Replacement transactions might not be accepted into the mempool due to the following reasons:

1. The transaction is not standard.
2. The transaction does not comply with the Replace-by-Fee policy, specifically the minimum feerate required.

The first problem is actually described in the `TODO` comment from the snippet below, extracted from the `resendSecondTxOfCheckpointToBTC` function. In case the required replacement transaction fee is greater than the change output value, the entire change output value is used as fees. This leaves the transaction with a 0-value (dust) output, causing the transaction to be considered as non-standard.

Additionally, since the replacement transaction fee can be capped to whatever the change output value is, this might cause the fee to fall below the minimum feerate required for the Replace-by-Fee policy. On top of that, Coinspect did not find additional validations of mempool replacement transactions such as making sure it pays an absolute fee of at least the sum paid by the original transactions, or that the additional fees covers the replacement transaction's bandwidth rate.

```
// set output value of the second tx to be the balance minus the bumped
fee
// if the bumped fee is higher than the balance, then set the bumped
fee to
// be equal to the balance to ensure the output value is not negative
balance := btcutil.Amount(tx2.Tx.TxOut[changePosition].Value)

// todo: revise this as this means we will end up with output with
value 0 that will be rejected by bitcoind as dust output.
if bumpedFee > balance {
        rl.logger.Debugf("the bumped fee %v Satoshis for the second tx
is more than UTXO amount %v Satoshis",
                bumpedFee, balance)
        bumpedFee = balance
}
```

Recommendation

Do not force the utilization of **all** the previously selected inputs to avoid capping the replacement transaction fee value. Instead, keep the input from the first checkpoint segment transaction and choose new inputs to account for the replacement transaction fee.

Make sure the replacement transaction follows the requirements listed in Bitcoin's Replace-by-Fee Policy.

Status

Fixed in pull request #232, #252 and #259. The function computes the required RBF fee, and the transaction is re-funded if necessary.

BP2-020

Slashing can be avoided with unbonding transactions

Status

Solved

✓

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Low

Location

`vigilante/btcstaking-tracker/btcslasher/slasher_utils.go`

Description

Any staker has a chance of avoiding slashing using the unbonding transaction.

The issue happens because the Vigilante fails to detect an unbonding transaction during the slashing process.

When a slashing event is triggered in Babylon, the slasher submits both the slashing and unbonding slashing transactions.

```
txHash1, err1 := bs.sendSlashingTx(fpBTCPK, extractedfpBTCSK, del,
false)
txHash2, err2 := bs.sendSlashingTx(fpBTCPK, extractedfpBTCSK, del,
true)
```

The issue arises through an implicit race condition. The BTC node connected to the `vigilante` may accept the slashing transaction and reject the unbonding slashing as it cannot accept both. However, the unbonding transaction (not slashing) may be already sent to other nodes on the network by the equivocating finality provider. In this case, the miner will accept the first transaction. In the event of such a transaction being the unbonding transaction, the vigilante will not send the unbonding slashing transaction, allowing the malicious finality provider to avoid being slashed.

Recommendation

Monitor the inclusion of the slashing transaction in a block and notify the maintainers if it is not detected within a specified timeframe. Note that this may require a database.

If an unbonding transaction is found in a Bitcoin block, check whether the unbonding slashing transaction must be sent.

Status

Fixed in pull request #209. When slashing, the `vigilante` will now wait for the confirmation of one of the two transactions sent. If neither is confirmed, it will attempt to send them again.

BP2-021

Lack of transaction retry mechanism may result in delayed transactions

Status

Solved

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
vigilante/submitter/relayer/relayer.go
```

Description

The Vigilante Submitter's relayer lacks a retry mechanism for error handling during Bitcoin transaction creation and submission, possibly delaying checkpoint transactions.

The Submitter will only re-attempt to submit the checkpoint once the configured `PollingIntervalSeconds` has elapsed.

Additionally, Coinspect identified several transaction validation checks in the `buildTxWithData` function that may be imposing unnecessary constraints on the transactions, which can cause the transaction's submission to fail.

The `buildTxWithData` function is responsible for creating the two transactions needed to submit Babylon checkpoints to Bitcoin. Below are some validation checks extracted from this function, which impose restrictions not strictly required by Bitcoin.

```
minRelayFee := rl.calcMinRelayFee(txSize)
if hasChange && changeAmount < minRelayFee {
    return nil, fmt.Errorf("the utxo value is insufficient for relaying
the transaction. Required: %v, Have: %v", minRelayFee, changeAmount)
}

// Ensuring the transaction fee does not exceed the utxo value
if hasChange && changeAmount < txFee {
    return nil, fmt.Errorf("the utxo value is insufficient for paying
the calculated transaction fee. Required: %v, Have: %v", txFee,
changeAmount)
}

// Ensuring change does not fall below the dust threshold
change := changeAmount - txFee
if hasChange && change < dustThreshold {
    return nil, fmt.Errorf("change amount %v is less than the dust
threshold %v", change, dustThreshold)
}
```

As an example, the final `if` block calculates a `change` variable by subtracting the transaction fee from the change amount and throws an error if this value is below the dust threshold. Additionally, the error message suggests a possible miscalculation of the change amount.

Recommendation

Implement a mechanism to retry Bitcoin transactions when errors occur. Alternatively, shorten the polling interval and provide clear documentation explaining the need for a reduced interval.

Review the transaction validations and remove those that are unnecessary. Otherwise, consider adding explanatory comments to clarify their intent. Consider enforcing only the validation requirements required by Bitcoin nodes for standard transactions.

Status

Fixed in pull request #229. The unnecessary checks were removed. The retry mechanism has not been made more aggressive, as there is no necessity for checkpoints to be included more quickly.

BP2-022

Single submitter can submit all checkpoints due to delayed sealed checkpoint processing

Status
Solved

✓

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
–

Location

`vigilante/submitter/submitter.go`

Description

A modified Vigilante submitter can submit `SEALED` checkpoints ahead of those using the standard version. This issue has no impact in the current Babylon system as there are no rewards for submitting checkpoints. Nevertheless, it must be taken into account if a reward system is put in place.

Currently, submitters only process the first pending `SEALED` checkpoint and do not submit the next one until Babylon updates its status to `SUBMITTED`. This allows an opportunistic submitter to get ahead by submitting all available `SEALED` checkpoints first.

The `PollSealedCheckpoints` function below retrieves `SEALED` checkpoints, returning only the oldest one:

```
func (pl *Poller) PollSealedCheckpoints() error {
        res, err :=
pl.querier.RawCheckpointList(checkpointingtypes.Sealed, nil)
        if err != nil {
                return err
        }
        sealedCheckpoints := res.RawCheckpoints

        if len(sealedCheckpoints) == 0 {
                return nil
        }

        // Ensure the oldest checkpoint is selected
        oldestCkpt := sealedCheckpoints[0]
        for _, ckpt := range sealedCheckpoints {
                if oldestCkpt.Ckpt.EpochNum > ckpt.Ckpt.EpochNum {
                        oldestCkpt = ckpt
                }
        }

        pl.rawCkptChan <- oldestCkpt

        return nil
}
```

This function runs at intervals defined by `PollingIntervalSeconds`, which defaults to 60 seconds unless overridden:

```
ticker := time.NewTicker(time.Duration(s.Cfg.PollingIntervalSeconds) *
time.Second)
for {
    select {
    case <-ticker.C:
        s.logger.Info("Polling sealed raw checkpoints...")
        err := s.poller.PollSealedCheckpoints()
...
```

As a result, submitters will only attempt to send the next checkpoint once the Vigilante reporter module confirms the previous one to Babylon.

Recommendation

Allow submitting of multiple `SEALED` checkpoints at once.

Status

Acknowledged. When reporting this issue, Babylon documentations described a system of rewards that would have been vulnerable by the problem described. Babylon Labs team has updated the documentation and stated rewards for checkpoint submitters will not be distributed in the near term.

BP2-023

Submitter transactions may not be relayed due to very low change output value

Status

Solved

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

–

Location

```
vigilante/submitter/relayer/relayer.go
```

Description

When a transaction is the first segment of a Babylon checkpoint and lacks a change output, the `submitter`'s relayer module enforces the addition of a 546-sat change output, which meets the current threshold for dust values. While this value is regarded as safe, there is a risk of such transactions being rejected if the dust threshold increases. Any transaction containing outputs deemed as dust will be classified as non-standard and will not be relayed.

The 546-sat dust threshold is derived from the default fee rate of 3000 sat/kvB, as outlined in the Bitcoin node's dust threshold calculation.

This default rate of 3 sat/byte is documented in the Bitcoin node, which serves as the reference value. Although many nodes adhere to this setting, there is no consensus requirement for them to do so. Consequently, nodes that enforce a higher fee rate may reject transactions with such low-value outputs.

This issue is currently classified as informational since the 3 sat/byte rate is widely adopted. However, it is prudent to account for the possibility of future limitations arising from an increased default fee rate.

Below is a code snippet from the `buildTxWithData` function, demonstrating the use of the minimum accepted output value for the change output:

```
dustThreshold  btcutil.Amount = 546
...
changeOutput := wire.NewTxOut(int64(dustThreshold), changePkScript)
rawTxResult.Transaction.AddTxOut(changeOutput)
```

Recommendation

Consider increasing the change output value to provide greater resilience against future changes in network fee requirements.

Status

Acknowledged. The Babylon Labs team is aware that the parameter might change and will provide updates to the `vigilante` in case the parameter is modified for the Bitcoin chains.

BP2-024

Well-positioned attacker can obtain critical signatures

Status

Solved

✓

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Low

Location

```
covenant-emulator
finality-provider
```

Description

An attacker well positioned in the network can observe a finality provider's private-key protecting passphrase and request signatures directly from the `eotsd manager`. A similar problem can be found in the communications between the `covenant-emulator` and the remote signer.

This puts the finality providers and covenant members' signatures at risk. Note that protecting the passphrases used or the secret keys is *not* enough if attackers can use the signing servers as oracles to sign arbitrary data.

To understand the issue, consider how the `finality-provider` repository works as a whole: it contains a `finality-provider` proper package and an EOTS Manager run by a daemon called the `eotsd`. While the finality provider proper is in charge of the logic to synchronizing with a Babylon node; the `eotsd` is supposed to handle the signing and generation of public randomness itself.

While this schema is sound in theory, in the current implementation the `eotsd` and the `finality-provider` both have access to the keyring and share a `passphrase` to unlock it. The `passphrase` is observable to an attacker positioned in the network: it is shared via an insecure transport between the two.

```
func NewEOTSManagerGRpcClient(remoteAddr string)
(*EOTSManagerGRpcClient, error) {
        conn, err := grpc.NewClient(remoteAddr,
grpc.WithTransportCredentials(insecure.NewCredentials()))
        if err != nil {
                return nil, fmt.Errorf("failed to build gRPC connection
to %s: %w", remoteAddr, err)
        }

...

// NewFinalityProviderServiceGRpcClient creates a new GRPC connection
with finality provider daemon.
func NewFinalityProviderServiceGRpcClient(remoteAddr string)
(*FinalityProviderServiceGRpcClient, func() error, error) {
        conn, err := grpc.NewClient(remoteAddr,
grpc.WithTransportCredentials(insecure.NewCredentials()))
        if err != nil {
                return nil, nil, fmt.Errorf("failed to build gRPC
connection to %s: %w", remoteAddr, err)
        }
```

The same architecture is used for the covenant committee signatures, which uses the `remote signer` as a black box to request signatures for delegations.

Another risk is that the programs that act as signature requesters (i.e: `finality-provider` and `covenant-emulator`) do not authenticate the connection to the blockchain node that triggers a request. This is another vector via which an attacker could request arbitrary signatures, as they could provide the signature requesters with fake events from the blockchain.

Recommendation

Allow operators of finality providers and committee members to have a secure configuration between all hosts involved in the signature process. All messages between the nodes, the signature requester and the signer should be authenticated and encrypted. Coinspect recommends that a two-pronged strategy is used: one for programs specifically made for the signing process that can support authentication at the application level, and another for blockchain nodes which are harder to modify to support safe channels.

- For communications between the `finality-provider` and the `eotsd manager`, and `covenant-emulator` and `covenant-signer`, Coinspect recommends supporting HMAC-based integrity checks with a shared secret between the hosts. Alternatively, authentication via message signatures can be used.
- For communications between the blockchain nodes and the `finality-provider` and `covenant-emulator`, Coinspect recommends setting up SSH tunneling.

For SSH Tunneling, the responsibility to implement it would fall entirely on the operators of the programs.

Additionally and as a defense in depth mechanism, HTTPs should be supported for communications whenever possible.

Status

Fixed in #337, which removes the passing of the `passphrase` via the network. Additionally, PRs #109 for the covenant emulator and #364 for the finality provider feature HMAC-based authentication for internal communications.

BP2-025

Finality provider can miss signature due to overflow

Status

Solved

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
babylon/crypto/eots/eots.go
```

Description

If the commitment generated when creating a finality signature via `signHash` overflows when considered modulo `order(secp256k1)`, the process will fail and the signature will not be generated. Future attempts will also fail as the process is entirely deterministic.

Note that the likelihood of this issue triggering is vanishingly small, around `1 / (2**256-1 - order(secpt256k1)`.

Recommendation

Due to the small likelihood of the issue triggering in practice, document this potential scenario so that the risk is known for finality providers.

Status

Acknowledged. Babylon Labs team will document the risk.

BP2-026

Stake can get stuck with transactions bigger than 8 kilobytes

Status

Solved

✓

Resolution

Fixed

Risk

Medium

Impact

Medium

Likelihood

Medium

Location

`covenant-emulator/covenant-signer/config/server.go`

Description

Staking transactions with a size bigger than 8 kilobytes will not be processed as the `RemoteSigner` caps the data it receives at 8 kilobytes. The signing request involves several complete transactions and keys which can easily surpass the 8 kilobytes limit:

```go
type SigningRequest struct {
        StakingTx                       *wire.MsgTx
        SlashingTx                      *wire.MsgTx
        UnbondingTx                     *wire.MsgTx
        SlashUnbondingTx                *wire.MsgTx
        StakingOutputIdx                uint32
        SlashingPkScriptPath            []byte
        StakingTxUnbondingPkScriptPath  []byte
        UnbondingTxSlashingPkScriptPath []byte
        FpEncKeys                       []*asig.EncryptionKey
}
```

Consider the loop in the `covenant-emulator` that uses a `signer` to sign the transactions. The signer implementation is an HTTP server that has set the `MaxContentLength` as 8 kilobytes:

```
func DefaultServerConfig() *ServerConfig {
        return &ServerConfig{
                Host:             "127.0.0.1",
                Port:             9791,
                WriteTimeout:     15,
                ReadTimeout:      15,
                IdleTimeout:      120,
                MaxContentLength: 8192,
        }
}
```

As such, `SigningRequest`s that do not fit the `MaxContentLength` of the `signer` will not get processed. If the delegator already included their transaction in Bitcoin, the stake would get stuck until it gets expired.

Recommendation

Increase the default `MaxContentLength` so that it is capable of processing all reasonable `SigningRequests`.

Because Bitcoin transactions have no defined size limit (except for the block size) and the `signer` communicates via a HTTP with a single, trusted, client, it is recommended to put a limit of 10Mb; which should be more than enough for the information carried in the request.

Status

Fixed in pull request #108.

BP2-027

Stake can get stuck due to race condition when delegating to a slashed finality provider

Status

Solved

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

–

Location

`babylon/crypto/eots/eots.go`

Description

Stake can get stuck when a new delegator includes their staking transaction into Bitcoin and calls `CreateBTCDelegation` while the finality provider they have delegated to is slashed.

This is due to the fact that delegating to a slashed finality provider will cause the `CreateBTCDelegation` transaction to fail on Babylon:

```
// 4. Check finality providers to which message delegate
// Ensure all finality providers are known to Babylon, are not slashed
for _, fpBTCPK := range
parsedMsg.FinalityProviderKeys.PublicKeysBbnFormat {
    // get this finality provider
    fp, err := ms.GetFinalityProvider(ctx, fpBTCPK)
    if err != nil {
        return nil, err
    }
    if fp.IsSlashed() {
        return nil, types.ErrFpAlreadySlashed.Wrapf("finality key: %s",
fpBTCPK.MarshalHex())
    }
}
```

Nevertheless, the delegator might have sent the Bitcoin transaction to delegate *before* the finality provider was slashed. If this happened, the delegator would get their stake locked by not fault of their own.

Recommendation

Document this risk so that delegators are aware of it.

Status

Fixed in PR #602. Warnings have been added to the documentation.

BP2-028

Change in parameters leads to slashing being missed

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Medium

Location

```
babylon/x/btcstaking/keeper/grpc_query.go
```

Description

A change in the `params.CovenantQuorum` necessary to consider a delegation as `active` will lead to missing slashings as the `vigilante` will use the `BTCDelegation` query to know the status of a delegation, but this query uses the current parameters to assert the status of the delegation instead of the delegation's parameters.

To further understand the issue, consider that the `vigilante` must be able to detect `unbonding` or other stake-spending transactions to be able to report them back to the Babylon chain. This process is done by the `reportUnbondingToBabylon` method in the `StakingEventWacher` of the `vigilante`.

```
active, err := sew.babylonNodeAdapter.IsDelegationActive(stakingTxHash)
if err != nil {
    return fmt.Errorf("error checking if delegation is active: %w",
err)
}
verified, err :=
sew.babylonNodeAdapter.IsDelegationVerified(stakingTxHash)
if err != nil {
    return fmt.Errorf("error checking if delegation is verified: %w",
err)
}
if !active && !verified {
    sew.logger.Debugf("cannot report unbonding. delegation for staking
tx %s is no longer active", stakingTxHash)

    return nil
}
...
```

Note that the process will not continue if the `delegation` is not `active` or `verified`. Both of the queries are done via the same underlying GRPC query to the Babylon node: `BTCDelegation`, implemented in `babylon/x/btcstaking/keeper/grpc_query.go`. In particular, the `status` is fetched via the `GetStatus` method, which takes a `covenantQuorum` parameter:

```
status := btcDel.GetStatus(
    k.btclcKeeper.GetTipInfo(ctx).Height,
    k.GetParams(ctx).CovenantQuorum,
)
```

Note that `GetParams(ctx)` will fetch the latest possible parameters available. `GetStatus` will return `PENDING` if the delegation's covenant signatures do not reach the passed quorum:

```
// we are still pending covenant quorum
if !d.HasCovenantQuorums(covenantQuorum) {
    return BTCDelegationStatus_PENDING
}

...

func (d *BTCDelegation) HasCovenantQuorums(quorum uint32) bool {
        return len(d.CovenantSigs) >= int(quorum) &&
d.BtcUndelegation.HasCovenantQuorums(quorum)
}
```

This is a discrepancy with regards to delegation activation according to the Babylon node, which will consider a delegation `active` as long as it reached the covenant signatures required by the parameters when the delegation was created, as seen in `AddBTCDelegationInclusionProof` or `AddCovenantSigs`, which fetch the parameters according to the parameters used to create the delegation.

```
// in AddBTCDelegationInclusionProof or AddCovenantSigs
btcDel, params, err := ms.getBTCDelWithParams(ctx, req.StakingTxHash)
```

```
func (ms msgServer) getBTCDelWithParams(
        ctx context.Context,
        stakingTxHash string) (*types.BTCDelegation, *types.Params,
error) {
        btcDel, err := ms.GetBTCDelegation(ctx, stakingTxHash)
    ...
        bsParams := ms.GetParamsByVersion(ctx, btcDel.ParamsVersion)
    ...
}
```

All in all, this means that a delegation can be considered `active` *by the node* when it reaches its `params.CovenantQuorum` signatures; but it can be considered `pending` *by the vigilante* if the `params.CovenantQuorum` at the time of the query has increased. This leads the `vigilante` to consider stake-spending transactions irrelevant if the `params.CovenantQuorum` is increased.

Note that to make matters worse, the `covenant-emulator` will not send further signatures if the delegations has received enough signatures *according to the params used during the delegation creation*, as seen in `covenant-emulator/covenant/covenant.go`:

```
// 2. the quorum is already achieved, skip sending more sigs
stakerPkHex :=
hex.EncodeToString(schnorr.SerializePubKey(btcDel.BtcPk))
if btcDel.HasCovenantQuorum(params.CovenantQuorum) {
    ce.logger.Error("covenant signatures already fulfilled",
        zap.String("staker_pk", stakerPkHex),
        zap.String("staking_tx_hex", btcDel.StakingTxHex),
    )
    continue
}
```

Recommendation

Make the GPRC query use the same logic than the consensus layer to check if a transaction is active. Namely, check with the `params` used during delegation creation.

Status

Fixed in pull request #512. The `BTCDelegation` query now uses the `params.ConvenatQuorum` at the point of creation of the delegation.

BP2-029

Attacker can avoid slashing in BTC by using wrapped messages

Status

Solved

✓

Resolution

Fixed

Risk

High

Impact

High

Likelihood

High

Location

`vigilante/btcstaking-tracker/btcslasher/slasher.go`

Description

An attacker can send their `AddFinalitySigMsg`s in a wrapped-message such as `authz.MsgExec` or `wasm.MsgExecuteContract` to avoid detection of the `vigilante` process that is in charge of monitoring calls to `AddFinalitySig` and slash finality providers that misbehave.

The root cause of the issue is that the query performed by the `btcslasher` module of the `vigilante` assumes that the calls to `AddFinalitySig` will be the `message.action` in the transaction's event:

```
messageActionName         = "/babylon.finality.v1.MsgAddFinalitySig"
...
queryName := fmt.Sprintf("tm.event = 'Tx' AND message.action='%s'",
messageActionName)
// subscribe to babylon fp slashing events
```

```
bs.finalitySigChan, startErr =
bs.BBNQuerier.Subscribe(txSubscriberName, queryName)
if startErr != nil {
    return
}
```

Nevertheless, wrapped messages have as the `message.action` the action of the topmost, outer message. For example, a message dispatched via a CosmosWasm contract has as `action` `"/cosmwasm.wasm.v1.MsgExecuteContract`, and a authorized message via the `authz` module has as its `action` `"/cosmos.authz.v1beta1.MsgExec"`.

This means the `btcslasher` will not send these wrapped messages to the `finalitySigChan`. This in turn avoids further processing, which should discover equivocations by the finality provider and a slashing transaction to be sent to Bitcoin via the `slashingEnforcer` method.

Recommendation

Do not rely on `message.action` as an indicator of the execution of certain messages. Instead, add custom events to the `AddFinalitySig` method and subscribe to those events. Events are emitted regardless of how the message was executed.

Status

Fixed in pull request #234. The strategy to detect misbehavior has been changed to polling. The `vigilante` now queries periodically the Babylon node for `evidence` of misbehavior.

BP2-030

Attacker can post fake stake in Babylon by forking Bitcoin

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Low

Location

`vigilante/btcstaking-tracker/btcslasher/slasher.go`

Description

The `DefaultConfirmationDepth` of the Babylon chain for Bitcoin is only `10`. With current network conditions and hashing prices, the cost of performing a double-spend against Babylon is between $1M and $10M USD. The Babylon chain has, as of February 2025, around $6B USD in TVL. This large asymmetry between the cost of performing an attack and the total value protected by Babylon makes it reasonable for a well-funded attacker to perform a double spend against Babylon.

The concrete attack would work as follows: an attacker with enough hashing power could present to Babylon a staking transaction and wait for it to be confirmed according to Babylon's light client. Then, they would post a fork to Bitcoin where they already spent that staking transaction. From Babylon's point of view, that delegation is active and valid and allows the finality provider to vote. On the other hand, the staker cannot be slashed: the staking transaction has been re-organized out of Bitcoin's mainchain.

The exact economic incentives at play are hard to predict, as an attacker can have several motivations to carry out the attack. The most direct is to perform a short against Babylon's tokens or in a prediction market. Note too that they need not recoup their expense only from Babylon: they could attack other vulnerable protocols as well and pool the profits of attacking all protocols in Bitcoin that are vulnerable.

The cost of the attack was calculated using the current Bitcoin hashrate (at around ~800M TH/s) and the nicehash average price for the `SHA256AsicBoos` miner (around 0.54 BTC/EH/day).

- Bitcoin hashrate: ~800 EH/s (source)
- Cost of 1 EH per hour: ~0.0225 BTC (source, SHA256AsicBoost)
- Cost of 400 EH per hour (enough for 50%): 400 * 0.0225 = 9 BTC

So for two hours (approximating 10 confirmations and some surplus time) the cost is 18 BTC. At a price hovering around ~100K per BTC, that is 1.8M USD. Crypto51 estimates around double the cost at ~4M USD.

Note that this estimation is rather lax: it assumes a two-hour window, when the attacker would need only 10 confirmations, which should be a window of around 100 minutes. On the other hand, the current total hash rate of Bitcoin is slightly higher, but can vary with time and experience drops. This calculation is also abstract because it does not take into account the *availability* of said hashing power. It is nevertheless useful for illustrative purposes.

It is also important to note that an attacker does not necessarily need 50% to be successful. Consider an attack from big Bitcoin pools against Babylon: The two biggest pools have around ~23% of hashing power (over last 2Y). If they combine their hashing power and get 46% of the total, they would have a 75% chance of successfully executing an attack against Babylon.

Recommendation

Increase the confirmation time for Bitcoin. While 10 is a safe-limit for most transactions, due to the incentives to attack a high-value chain the limits need to be more conservative.

This script can be used to calculate the amount of confirmations given a risk tolerance. With `q` being the amount of hashing power an attacker can control and `z` is the amount of confirmations that are required.

```
>>> def attacker_success_probability(q, z):
...     p = 1.0 - q
...     lambd = z * (q / p)
...     s = 1.0
...     for k in range(z + 1):
...         poisson = math.exp(-lambd)
...         for i in range(1, k + 1):
...             poisson *= lambd / i
...         s -= poisson * (1 - (q / p) ** (z - k))
...     return s
```

Status

Babylon stated that for their mainchain the `ConfirmationDepth` parameter will be set to 30.

BP2-031

Chain bloated by checkpoints that are never forgotten

Status

Solved

✓

Resolution

Fixed

Risk

Low

▼

🟥 □ □

Impact

Low

Likelihood

Low

Location

`babylon/x/checkpointing/keeper/keeper.go`

Description

The `SetCheckpointForgotten` method does not call the `AfterRawCheckpointForgotten` hook. This in turn prevents the pruning of data from the `monitor`'s store of checkpoints, causing blockchain bloat.

The issue is in the `SetCheckpointForgotten` implementation, which never calls the hook. Compare this to the implementation of `SetCheckpointFinalized`, which does:

```
// invoke hook, which is currently subscribed by ZoneConcierge
if err := k.AfterRawCheckpointFinalized(ctx, epoch); err != nil {
    k.Logger(sdkCtx).Error("failed to trigger checkpoint finalized hook
for epoch %v: %v", ckpt.Ckpt.EpochNum, err)
}
```

Recommendation

Add a call to `AfterRawCheckpointForgotten` in the `SetCheckpointForgotten` method.

Status

Fixed in pull request #540. The hook is now called.

BP2-032

Users may have their transactions rejected due to the generation of dust outputs

Status

Solved

✓

Resolution

Fixed

Risk

Low

▼
🟥□□

Impact

Low

Likelihood

Low

Location

`babylon-wallet-connector/src/core/utils/mempool.ts`

Description

Users may waste time if their transactions are rejected by nodes due to non-standard formatting. This happens because the library does not ensure proper handling of dust change outputs, potentially generating non-standard transactions.

The `getFundingUTXOs` function is designed to select the minimum number of inputs required to meet a specified amount. However, it does not verify whether the excess amount (change) surpasses the dust threshold. As a result, transactions containing dust outputs are classified as non-standard and will be rejected by most nodes.

```
if (amount) {
    let sum = 0;
    let i;
    for (i = 0; i < confirmedUTXOs.length; ++i) {
        sum += confirmedUTXOs[i].value;
        if (sum > amount) {
            break;
        }
    }
    if (sum < amount) {
        return [];
    }
    sliced = confirmedUTXOs.slice(0, i + 1);
}
```

Coinspect was unable to determine where this function is executed, making it difficult to assess the exact impact.

Recommendation

Ensure that the resulting change does not fall below the dust threshold. Additionally, implement checks to confirm that the generated transaction adheres to the standard transaction policy.

Status

Fixed in PR #264. The `mempool` component was removed.

BP2-033

Unchecked user input allows retrieving wrong global parameters

Status

Solved

✓

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

-

Location

`btc-staker/cmd/stakercli/daemon/daemoncommands.go:401`

`btc-staker/cmd/stakercli/transaction/transactions.go:695`

Description

Users can provide an incorrect block height for the staking transaction, which can lead to the staker CLI retrieving the wrong versioned global parameters. This may result in crafting invalid requests or transactions that will be rejected by the nodes.

The `stakeFromPhase1TxBTC` and `createPhase1UnbondingTransaction` functions allow users to specify the block height at which the staking transaction was included. Below is a snippet from the `stakeFromPhase1TxBTC` function:

```
blockHeighTxInclusion := ctx.Uint64(txInclusionHeightFlag)
if blockHeighTxInclusion == 0 {
        resp, err := client.BtcTxDetails(sctx, stakingTransactionHash)
        if err != nil {
                return fmt.Errorf("error to get btc tx and block data
from staking tx %s: %w", stakingTransactionHash, err)
        }

        blockHeighTxInclusion = uint64(resp.Blk.Height)
}
```

This block height is then used to fetch the versioned global parameters needed for creating the delegation request or unbonding transaction:

```
paramsForHeight :=
globalParams.GetVersionedGlobalParamsByHeight(blockHeighTxInclusion)
if paramsForHeight == nil {
    return fmt.Errorf("error getting param version from global params
%s with height %d", inputGlobalParamsFilePath, blockHeighTxInclusion)
}
```

Therefore, since the CLI fails to validate the provided block height, providing an incorrect block height would cause the CLI to fetch the wrong global parameters, leading to crafting invalid requests or transactions.

Recommendation

Instead of relying on user-provided input, directly retrieve the transaction's block height from the node to prevent errors.

Status

Fixed in PR #160. The staking transaction height is no longer mandatory, making the default safe.

BP2-034

Fixed request ID in Keystone wallet requests prevents request-response association

Status

Solved

✓

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

–

Location

`babylon-wallet-connector/src/core/wallets/btc/keystone/provider.ts`

Description

The wallet connector library uses a fixed `requestId` for signature requests to Keystone wallets, preventing proper request-response association.

The snippet below, from the `signMessage` function, shows the usage of a fixed `requestId` and `origin` for every request.

```
const ur = this.dataSdk.btc.generateSignRequest({
    requestId: "7afd5e09-9267-43fb-a02e-08c4a09417ec",
    signData: Buffer.from(message, "utf-8").toString("hex"),
    dataType: KeystoneBitcoinSDK.DataType.message,
        {
        },
    ],
});
```

The requestId is intended to link responses to their corresponding requests. However, since the code does not verify whether the response matches the original requestId, messages and signatures cannot be reliably associated.

Additionally, accepting and trusting arbitrary origin values is a poor security practice, as they can be spoofed. Coinspect was unable to find Keystone's documentation on origin handling and did not have a Keystone wallet device to test potential exploits. Ensuring the security of wallets integrating with Babylon is outside the scope of this project.

Recommendation

Use a unique requestId for each signing request and validate that request and response IDs match.

Assess the impact of using an arbitrary origin in Keystone wallet signature requests.

Status

Fixed in PR #264. The ID is now a UUIDv4.

BP2-035

Malicious external API can crash staking service

Status

Solved

✓

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

-

Location

`staking-api-service/internal/shared/http/client/http_client.go`

Description

The `staking-api-service` HTTP client does not set a limit to the amount of data read from the host it is sending to request to. This means that if the host is malicious, it can attempt to crash the API by sending a big amount of data and fill the memory of the system running the Staking API.

The `sendRequest` method uses the subjacent's `net/http` Client `Do` method to make requests, but fails to limit the read from `resp.Body`, instead relying only on a timeout to close the connection.

```
resp, err := client.GetHttpClient().Do(req)
...
if err := json.NewDecoder(resp.Body).Decode(&output); err != nil {
```

A malicious API can stream as much data as allowed before the timeout expires. This data will be held in memory. If the API is reached on-demand via one the Staking API routers, like the `ordinals` API is, a malicious operator of the API can start an arbitrary number of connections that all stream a big amount of data at the same time, eventually leading to a Staking API crash.

Recommendation

Use `io.LimitReader` on the body before passing it to `json.NewDecoder`.

Status

Fixed in PR #246. An `io.LimitReader` of 10 megabytes max will be passed to the decoder.

BP2-036

Inability to replay a single unprocessable message can prevent API server from starting

Status

Solved

✓

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

`staking-api-service/cmd/staking-api-service/scripts/replay_unprocessed_messages.go`

Description

Coinspect observed that an error during the replay of unprocessable messages at server startup will prevent the server from launching.

The API service provides a mechanism to replay stored `UnprocessableMessages` into the queues. If the server is set to start with the `replayFlag` option, it will run the following function during server startup. Should an error occur, the function returns an error and immediately stops the reprocessing of pending messages.

```
func ReplayUnprocessableMessages(ctx context.Context, cfg
*config.Config, queues *v2queue.Queues, db dbclient.DBClient) (err
```

```
error) {
    ...
        // Process each unprocessable message
        for _, msg := range unprocessableMessages {
                var genericEvent GenericEvent
                if err := json.Unmarshal([]byte(msg.MessageBody),
&genericEvent); err != nil {
                        return errors.New("failed to unmarshal event
message")
                }

                // Process the event message
                if err := processEventMessage(ctx, queues,
genericEvent, msg.MessageBody); err != nil {
                        return errors.New("failed to process message")
                }

                // Delete the processed message from the database
                if err := db.DeleteUnprocessableMessage(ctx,
msg.Receipt); err != nil {
                        return errors.New("failed to delete
unprocessable message")
                }
        }
```

This error is caught by the following snippet in the `main` function at `cmd/staking-api-service/main.go`, which will immediately halt the server's operations.

```
if cli.GetReplayFlag() {
    log.Info().Msg("Replay flag is set. Starting replay of
unprocessable messages.")

    err := scripts.ReplayUnprocessableMessages(ctx, cfg, v2queues,
dbClients.SharedDBClient)
    if err != nil {
        log.Fatal().Err(err).Msg("error while replaying unprocessable
messages")
    }
    return
}
```

Since most API servers are configured to run unattended, those set to replay unprocessable messages are likely to attempt it again, causing continuous crashes until a human operator intervenes and configures the server to run without these options or clears conflictive unprocessable messages in the database.

Recommendation

The server should continue its operations regardless of the outcome of the `ReplayUnprocessableMessages` function.

Status

Acknowledged. The Babylon Labs team stated that they were not planning to run the script while the server is also functioning.

BP2-037

Generated protobuf code cannot be checked for integrity

Status

Solved

Risk

None

Resolution

Fixed

Impact

Recommendation

Likelihood

–

Location

`babylon-proto-ts`

Description

The contents of the `babylon-proto-ts` repository cannot be recreated with the given scripts found in the repository. This means that users cannot double check the integrity of the generated files. An attacker can leverage this seemingly auto-generated code to hide exploits. See the attack on XZ Utils for an example.

Recommendation

Specify a commit in `babylon-proto-ts` scripts so that the code can be regenerated and matched for integrity checks.

Status

Fixed in PR #22. The project now establishes the precise Babylon commit from which the files are generated, making integrity checks viable.

BP2-038

Bitcoin transactions might be rejected due to insufficient minimum feerate

Status

Solved

Resolution

Fixed

Risk

Low

Impact

Low

Likelihood

Low

Location

```
btc-staking-ts/src/utils/staking/index.ts 241:245
btc-staking-ts/src/utils/fee/index.ts
```

Description

Transactions created with the `btc-staking-ts` library are at risk of not being relayed by nodes if the feerate provided is too low.

The snippet below shows the `getWithdrawTxFee` function, which adds a buffer fee function to the base fee.

```
export const getWithdrawTxFee = (feeRate: number): number => {
  const inputSize = P2TR_INPUT_SIZE;
  const outputSize = getEstimatedChangeOutputSize();
  return (
    feeRate *
      (inputSize +
        outputSize +
        TX_BUFFER_SIZE_OVERHEAD +
        WITHDRAW_TX_BUFFER_SIZE) +
    rateBasedTxBufferFee(feeRate)
  );
};
```

Such a buffer fee is calculated by the `rateBasedTxBufferFee` as shown below. If the feerate falls below the minimum relay rate accepted by most nodes (3 sats/vbyte), this function returns a fixed value of 30 to offset the potential fee shortfall.

```
const rateBasedTxBufferFee = (feeRate: number): number => {
  return feeRate <= WALLET_RELAY_FEE_RATE_THRESHOLD
    ? LOW_RATE_ESTIMATION_ACCURACY_BUFFER
    : 0;
};
```

However, this buffer may not adequately cover the minimum relay feerate required by most nodes, which could lead to the transaction being rejected. A similar issue exists in the `getStakingTxInputUTXOsAndFees` function.

In contrast, the `validateStakingTxInputData` function only performs minimal validation of the `feeRate` parameter, without ensuring it meets the minimum standard relay rate enforced by most nodes.

```
if (feeRate <= 0) {
    throw new StakingError(
        StakingErrorCode.INVALID_INPUT, "Invalid fee rate",
    );
}
```

Recommendation

Require a `feeRate` of at least 3 sats/vbyte instead of relying on a potentially inadequate buffer.

Status

Fixed in PR #69. The Babylon Labs team added warnings for users of the library not to set an invalid `feeRate`. The Babylon Labs team prioritizes flexibility for users of the library, so it was not desired to limit or reject `feeRate`s that were too low.

Users need to be aware of the implications of setting their `feeRate`.

BP2-039

Library users can create invalid staking transactions

Status

Solved

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
btc-staking-ts/src/staking/stakingScript.ts
```

Description

Users of the `btc-staking-ts` can create staking transactions that will be rejected by Babylon. This is due to the fact that the exported `StakingScriptData` class supports building the staking script with the public key of different finality providers. Nevertheless, this feature is not supported by the Babylon node.

The mismatch can be observed in the code of `StakingScriptData`. Its constructor takes a list of `finalityProviderKeys`:

```
constructor(
    ...
    // A list of public keys without the coordinate bytes corresponding
to the finality providers
    // the stake will be delegated to.
    // Currently, Babylon does not support restaking, so this should
contain only a single item.
    finalityProviderKeys: Buffer[],
    ...
) {
```

Note that although it is documented that this should only have a single item, this invariant is not checked in the library.

If a user mistakenly uses the library with a `finalityProviderKeys` list of more than one element, their generated scripts will create invalid staking transactions, leading to value locked in Bitcoin for the end users.

Recommendation

Generate a runtime error for arrays of length bigger than one until restaking is enabled at the chain level.

Status

The Babylon Labs team acknowledged the issue but stated that the main interface for creating transactions will be `manager.ts` and this will be documented in the `README` of the project.

BP2-040

Attackers can eclipse node and post a fake Bitcoin chain

Status

Solved

✓

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Description

A Babylon node that is syncing to the network is vulnerable to being eclipsed and fed a Bitcoin chain that is not the canonical. This attack assumes the attacker can eclipse the victim. A victim that has access to at least one honest Babylon peer is not affected directly, as their node will `panic` upon observing a fork.

Recommendation

A `CheckConsistency` method can be added to the CLI of the node. This tool would accept a Bitcoin header hash and a height and would cross-check against the Babylon light client. This tool can be used as the documented process to bootstrap a node.

`CheckConsistency` should `panic` the node and print a warning if the node light client does not match the Bitcoin header provided by the user.

Status

Acknowledged. Babylon Labs team has stated that a possible mitigation for users is to run the `monitor` program while syncing their node and use the logs to reach the same conclusion as the `CheckConsistency` method would provide.

They are working on documentation that explains this risk to operators and how to mitigate it exactly via the `monitor` program.

BP2-041

Malicious indexer instance can crash vigilante staking tracker

Status

Solved

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

-

Location

`btcstaking-tracker/indexer/indexer.go`

Description

A malicious indexer can crash the vigilante staking tracker by sending an excessive amount of data. Since there's no limit on the data read from the indexer, this could eventually consume all of the host's memory.

The root cause of this issue was initially described in `BP2-035`. As shown below, the `GetOutspend` method fails to restrict how much data is read from `resp.Body`:

```
resp, err := c.httpClient.Do(req)
...
return json.NewDecoder(resp.Body).Decode(&response)
```

While this service is intended for local usen —making exploitation unlikely— a vigilante might opt to use an untrusted third party.

Recommendation

Apply `io.LimitReader` to the body before passing it to `json.NewDecoder`.

Status

Fixed in PR #241.

BP2-042

BLS private key exposed at rest

Status

Solved

✓

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

–

Location

```
app/signer/private.go
```

Description

During the review of the BLS keystore pull request, Coinspect observed that the BLS private key remains recoverable from the host itself or a data-backup of it due to inadequate storage segregation between the key and the associated passphrase.

Currently, the BLS key is encrypted using a passphrase-derived key provided by the user and stored on the host. However, the passphrase itself—potentially even an empty password—is also saved in plaintext in a separate file. While this setup may prevent an attacker with access only to raw storage data from recovering the private key, it fails to protect against an adversary who restores the host's filesystem.

Furthermore, Coinspect identified multiple instances in the codebase indicating that BLS keys are still referenced within the `priv_validator_key.json` file, including `x/checkpointing/client/cli/tx.go` and `cmd/babylond/cmd/create_bls_key.go`.

Recommendation

Inject the passphrase via an environment variable so that it remains in memory, thereby segregating it from the private key it is meant to protect. Consider preventing the usage of empty or easy-to-guess passwords.

Adjust mentions to BLS keys held inside the `priv_validator_key.json` file accordingly.

Status

Acknowledged. Babylon Labs team is working on a mitigation for these problems for future releases.

BP2-043

Lack of differential testing between implementations of Bitcoin script libraries

Status

Solved

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

-

Location

```
btc-staking-ts
babylon/btc-staking
```

Description

Both the `babylon/btc-staking` and the `btc-staking-ts` libraries function as helper libraries to create and validate Babylon-related Bitcoin transactions. Nevertheless, none of these projects make use of differential testing to find potential differences in their implementations.

Recommendation

Improve the test suite by adding differential testing. One possible implementation is:

- generate random inputs
- feed those inputs to both `btc-staker`, `babylon/btcstaking` and `btc-staking-ts`
- check that the scripts or transactions match for the same set of inputs

Status

The Babylon Labs team acknowledged this recommendation and will work on improving the test suite with differential testing.

BP2-044

Proof-of-Possession (PoP) allows for reuse

Status

Solved

✓

Resolution

Acknowledged

Risk

None

Impact

Recommendation

Likelihood

–

Location

```
babylon/x/btcstaking/types/pop.go
```

Description

Since PoPs only include a Babylon address and are not tied to a specific timestamp, application, or use, they could potentially be replayed or reused. The current safeguard against this is enforcement within Babylon's node code, such as authentication checks for the staker address and ensuring the staking transaction is not submitted twice.

Additionally, Babylon supports ECDSA signing for Legacy (P2PKH) addresses. While BIP-322 acknowledges it as a valid option, it recommends transitioning to the new PoP signing format for all address types.

Recommendation

Ideally, implement a mechanism that allows users to sign PoPs using clear-text signing instead of signing a hash. This would enable users to review the exact content their wallets are signing, improving transparency. As a reference, consider the Sign-in with Ethereum (SIWE) implementation, which enables users to prove ownership of their EVM address by signing a clear-text message.

Additionally, incorporate a nonce and a Babylon-specific tag in PoP generation to prevent reuse within Babylon or by third parties.

Status

Acknowledged. The Babylon team is aware of the limitations of the current PoP signing mechanism.

6. Disclaimer

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

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

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

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

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