Security Audit Report

Babylon Q2 2025: Genesis v2 Upgrade

Last Revised
2025/06/06

Authors:
Karolos Antoniadis, Mirel Dalcekovic, Ivan Gavran, Martin Hutle, Aleksandar Ljahovic, Andrija Mitrovic

Contents

Audit overview

The Project

In May 2025, Babylon has engaged Informal Systems↗ to conduct a security audit of an upgrade with key features aimed at delivering significant quality-of-life improvements as part of the Babylon Genesis v2 Upgrade. The consultants will provide auditing services for the changes introduced in this upgrade, which have been categorized by priority into four classes, as agreed with the Babylon team.

Scope of this report

The scope of this audit is limited to the changes introduced in the pull requests listed in detail in the ***Audit Dashboard*** section of this report. Our review and threat modeling efforts are focused exclusively on this new or modified logic. While our analysis centers on the specific code diffs, we have considered the broader context of the Babylon protocol to evaluate how these changes may affect the system as a whole.

The pre-existing implementation is considered correct and out of scope for this audit; however, some code outside of the listed PRs was reviewed as necessary to support our threat modeling efforts, though this was not the primary focus of our review.

The newly introduced changes were categorized into four classes based on their significance and potential impact on the protocol:

- **Class 1 – High Priority Functional Changes:**

  Includes the reward distribution fix, unlocking of the EOTSD keyring feature, and optimizations in the Vigilante BTC staking tracker.

- **Class 2 – Medium Priority Functional Changes:**

  Covers integration of the Token Factory module, PFM, and IBC middleware. The review focuses on wiring correctness and potential negative impacts on the Babylon Protocol, including modifications to Genesis logic.

- **Class 3 – Low Priority Fixes:**

  Minor bug fixes and adjustments.

- **Class 4 – Non-Functional Changes:**

  Includes refactoring, comments, formatting, and other non-functional updates.

The consultants will conduct a threat model, threat analysis, and comprehensive code inspection for changes classified under Class 1 and Class 2. For Class 3 and Class 4, the audit scope is limited to a comprehensive code inspection.

Audit plan

The audit was conducted between May 19, 2025 and June 3, 2025 by the following personnel:

- Martin Hutle

- Ivan Gavran
- Karolos Antoniadis
- Aleksandar Ljahović
- Andrija Mitrović
- Mirel Dalčeković

Conclusions

The auditing team's conclusions regarding protocol design and code implementation quality are organized by scope and by categories of PRs reviewed during this audit.

The changes to the reward distribution module were adding a simple mechanism to a complex existing code. Our analysis was thus focused on checking that the changes that were introduced indeed implement the change of semantics that was intended, and that the new mechanism provides the right inputs to the existing computation. There was no findings surfaced in the code, which seems to be well written and robust.

In regards to the "Unlocking EOTSD keyring" scope: The code is well written and the move to a file-backend keyring is meaningful. Besides minor informational findings, we did not find any issue. However, future changes to `cosmos/crypto/keyring`↗ or `cosmos/keyring`↗ may introduce compatibility issues for the EOTS manager, so care needs to be taken when upgrading those dependencies.

The BTC staking tracker *Vigilante* was audited following recent optimization changes. The audit identified a few minor issues, along with one notable bug in the periodic block-fetching logic, where the condition for fetched blocks was incorrect. Additionally, a race condition - unlikely but possible - was discovered. Overall, the *Vigilante* tracker is considered a robustly designed and implemented off-chain component. It reliably handles delegation status updates and avoids unintended duplicate actions on the Babylon chain, thanks to consistent validation of delegation status transitions.

The pull requests, grouped into genesis init/export logic, small fixes, and non-functional changes, were overall well-implemented and aligned with the goals of the v2 backport effort. While most of the changes appear solid and raise no major concerns, a few areas were identified where improvements could be made - primarily around edge case handling and input validation.

All identified findings are documented in the ***Findings*** section of this report.

Audit Dashboard

Target Summary

- **Type:** Protocol & Implementation
- **Platform:** Go
- **Artifacts, organized per class:** - **Artifacts, organized per class:** The audit will be based on the analysis of:
  - commit hash `d95f863`↗ in the babylon repository↗, with a focus on pull requests introducing changes as part of the new **Genesis v2 Upgrade.**
  - commit hash `5da4342`↗ in the finality-proider repository↗ with a focus on the: Unlocking eotsd keyring PR↗.
  - commit hash `33ba9b4`↗ in the vigilante repository↗ with a focus on the: Optimization in Vigilante BTC staking tracker PR↗

  Pull requests containing changes were classified in:

  - **Class 1 - critical features:**
    - → Reward distribution fix PR↗ - fix introduced in the babylon repository↗.
    - → Unlocking eotsd keyring PR↗ - feature introduced in the finality-provider repository↗.
    - → Optimization in Vigilante BTC staking tracker PR↗ - optimization introduced in the vigilante repository↗.
  - **Class 2 - integrations and genesis PRs:**
    - → Babylon's fork of Strangelove's token factory↗ module integration PR↗
    - → IBC middlewares integration
      - · IBC callbacks↗
      - · PFM integration PR↗
      - · rate limiter↗
      - · ICA and ICQ modules↗
    - → Genesis logic PRs:
      - · Backport: chore(x/btccheckpoint): add import/export genesis logic↗
      - · Backport: chore(x/btcstaking): add AllowedStakingTxHashes and LargetsBTCReorg to import/export genesis logic↗
      - · Backport: chore(x/epoching): add init/export genesis logic↗
      - · Backport: chore(x/finality): Update init/export genesis logic↗
      - · backport: chore(x/mint): update gen logic↗
      - · [backport] chore(x/monitor): Add init/export genesis logic↗
  - **Class 3 - small fixes PRs:**
    - → [release/v2.x] add v1.1 upgrade data↗
    - → backport: v2x fix inactive fp signing info↗
    - → chore: backport #850 BLS improvement including permission and and verification↗
    - → backport(bugfix): add checks for slashed fp in gov resume finality↗
    - → [backport] chore: add bank wrapper for distribution mod account↗
      - · Security fix↗; review that the addition of token factory doesn't have similar impact as reported in SA
    - → [backport] chore: Add ValidateBasic for CommitPubRandList↗
    - → [backport] fix: update tokenfactory params to use ubbn as fee denom↗

- **Class 4: non-functional changes**
  - [backport] chore: upgrade the make file: release and build↗
  - fix: wire btcstaking to btc light client hooks↗
  - [backport] chore: Add e2e test for v2 upgrade↗
  - fix: remove protobuf from make file↗
  - [Backport v2] chore: move goreleaser job to release workflow↗
  - [release/v2.x] chore: ensure release workflow build the released tag↗
  - [release/v2.x] chore: Add btcstaking with new testing framework↗
  - [release/v2.x] fix: btcstaking test fix↗
  - [backport] chore(x/ibc-fee): remove deprecated mod↗
  - [backport]1.1 Changelog to v2.x↗
  - fix: missing proto version bumps↗

Engagement Summary

- **Dates**: May 19, 2025 - June 3, 2025
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	Number
Critical	0
High	0
Medium	5
Low	4
Informational	5
Total	14

System Overview

The main goal of Babylon's Genesis v2 upgrade is not to introduce new Babylon protocol features, but to **enhance cross-chain composability** and prepare the chain for broader ecosystem integration.

Key additions include **IBC Callbacks**, **Packet Forwarding Middleware (PFM)**, **Token Factory**, and **Rate Limiting**. These features are mostly imports of established modules from the broader Cosmos ecosystem, but were carefully delayed until now due to their **wide-reaching implications** on IBC behavior and security. While the features are not yet in use, **projects building on Babylon are actively waiting** for them and are expected to start integrating them soon after the upgrade.

The upgrade includes several critical changes: a **fix for rewards distribution**, **optimizations** to the **Vigilante BTC staking tracker**, and support for **unlocking the EOTSD keyring** -enabling keys to be stored in an encrypted file and unlocked with a password.

Additional changes include the introduction of genesis logic, non-functional improvements, and minor fixes related to the upgrade and backported bug fixes.

Each of the critical areas will be described in a dedicated system overview section below.

Integration scope

The integration scope includes additions of following modules:

- Packet Forward Middleware (PFM)
- TokenFactory
- IBC Callbacks
- Rate Limiter
- ICA and ICQ

And also the removal of ibc-fee module.

In this release, the modules were only enabled to be used, but without introducing new features using them.

Rewards distribution fix

The reward distribution module uses a mechanism similar to `x/distribution` to **allocate** the delegator's rewards to the finality providers and then **distribute** these rewards to the delegators once the delegation changes.

The basic change of the PR under audit is that previously the wrong distribution table was used. If rewards for height *h* are distributed later at some height *h'*, the distribution table and voting information of height *h* (and not *h'*) should be used.

This is implemented by tracking all delegation changes for *h* in `BeginBlocker`, and when rewards are distributed at height *h'* in `EndBlocker`, the tracked information is fed into the existing reward mechanism.

The audit is using a threat model that covers the whole mechanism, while the inspection was focusing on the changes, assuming that the calculation mechanism itself was audited before. The basic properties are therefore that the input to these calculations is correct with respect to height and order.

Event based BTC staking tracker

The new, optimized BTC staking tracker introduces an event-based monitoring system that significantly **optimizes how Vigilante tracks delegation state changes on Babylon**. Previously, Vigilante inefficiently polled *all* pending delegations from Babylon on a recurring basis, resulting in substantial data overhead -reportedly around 20TB per month on testnet, according to the Babylon team.

The new implementation adopts an **event-driven model**: rather than polling all delegations, Vigilante now listens to CometBFT block events (e.g., delegation activation events) and updates its internal state incrementally on a block-by-block basis. This allows it to track only the relevant delegations and subsequently fetch related data from the Bitcoin side.

While this approach improves performance and scalability, it introduces potential failure modes. If Vigilante misses or misprocesses a key event, it may fail to track specific delegations, leading to incorrect behavior.

Vigilante maintains two in-memory tracking structures `-pendingTracker` and `unbondingTracker-` to monitor active and pending delegations. These structures are then used by the BTC staking tracker component to determine which unbonding messages to send to Babylon. Ensuring reliable event processing and accurate population of these tracking structures is now critical to correctness, as failures could result in missed activation and unbonding of delegations.

Evaluating the correctness and completeness of this delegation-tracking logic -and identifying any scenarios where delegations may be missed- was one of the primary focuses of this security audit.

Unlocking EOTSD keyring

Up until this version, the EOTSD keyring has been using the test↗ keyring backend, where keys are stored unencrypted on disk (and is thus unsuitable for production).

This release changes the keyring backend into the production-ready file backend↗. As a part of this change, support for reading a passphrase from the user, unlocking with the passphrase, and storing keys in an in-memory map needed to be implemented.

Small fixes, genesis and non functional changes

Several smaller backported changes were introduced to improve correctness, validation, and developer experience across modules. These include fixes for inconsistent finality provider signing info, enhanced BLS signature handling with proper permission checks and verification, validation logic for governance messages, updates to token factory parameters, and utility improvements such as a bank wrapper for the distribution module.

As part of the v2 upgrade backporting effort, multiple modules (`x/btccheckpoint`, `x/btcstaking`, `x/epoching`, `x/finality`, `x/mint`, and `x/monitor`) have been updated to implement or extend their `InitGenesis` and `ExportGenesis` logic. These changes ensure that all relevant module states are correctly initialized and persisted. In addition to standard serialization, several modules also introduced or improved validation of genesis fields to maintain consistency and prevent runtime issues during chain initialization.

Several non-functional changes were introduced as part of the v2 backport process to improve development workflows, testing infrastructure, and release consistency. These include version bumps, Makefile updates and cleanup, relocation of release and GoReleaser jobs to dedicated workflows, and addition of an E2E upgrade test for the v2 transition.

Threat Model

Our threat analysis begins by defining a set of properties essential for the correctness of the audited scope. For each property, we identify one or more associated threats and analyze the conditions under which they might be violated.

The results of this analysis are presented in the ***Findings*** section.

Event based BTC staking tracker threat model

BTC staking tracker Vigilante's failure to capture or process an event - whether due to bugs in event tracking or delegation tracking on vigilante, or due to missed blocks - can result in **incomplete or incorrect delegation state tracking**. Such issues can cause critical operational failures, such as skipped delegation activation, duplicated or missing undelegation processing.

This threat model focuses on identifying potential failure modes introduced by the recent PR↗ containing optimizations to Vigilante's delegation retrieval logic. To thoroughly assess correctness the analysis slightly extended beyond the immediate pull request scope. Our objectives were to:

- Understand the BTC staking tracker implementation.
- Compare the original and event-based, optimized Babylon delegations processing logic on Vigilante.
- Understand delegation status transitions on Babylon and the corresponding event emissions (as implemented in `x/btcstaking` and `x/finality`), in order to determine whether the original implementation can serve as a reliable reference point for expected behavior.
- Identify whether any new logic introduces inconsistencies or risks.

Genesis related PRs threat inspection

In our correctness analysis of the genesis proto definitions, `InitGenesis`, `ExportGenesis` and `ValidateGenesis` PRs, we focused on verifying the following:

- The changes introduced by the pull requests properly update the module's `InitGenesis` and `ExportGenesis` logic to ensure the module's state can be fully and accurately imported and exported. This includes correctly handling any newly added or previously missing genesis fields in a consistent manner.
- Since only a properly formatted and valid JSON file can be used to initialize or restore module state, we reviewed whether the genesis proto definitions use the same field types as other proto files referring to the same data - ensuring consistency and compatibility.
- `InitGenesis` can be invoked during chain initialization or during a chain migration (i.e., hard fork) using the genesis file provided by the operator. The resulting state must be validated by calling `ValidateGenesis`. We specifically analyzed the validation of externally injected data that becomes part of the module's genesis state and verified that the same validation logic is applied during genesis initialization. This ensures consistency and prevents invalid data from entering the blockchain state at startup.

Consequences of incorrectly implemented genesis functionality

Improper implementation of `InitGenesis`, `ExportGenesis`, or associated validation logic can lead to several critical issues:

- Inability to export or re-initialize state: If the current state of the Babylon blockchain cannot be accurately exported and re-imported, there is a risk of data loss. The inability to create reliable backups makes the system fragile, limits operational flexibility, and complicates long-term maintenance.
- Inconsistent validation logic: If validation logic is missing or misaligned between `ValidateGenesis` and `ValidateBasic` functions, then during a hard fork/ migration, the `InitGenesis` call may accept malformed or invalid genesis JSON. This can result in:
  - Chain halts due to unexpected state errors or
  - Initialization of an invalid or corrupted state in the store, which may be hard to detect or recover from.

Rewards distribution, vigilante, EOTSD manager

Property EOTSD-01: The private key is only loaded in memory if the correct passphrase is provided

Conclusion

The `Unlock`↗ of the private key (i.e., the loading of the key in memory↗) calls `getKeyFromKeyring` that uses the provided `passphrase`. Inside `getKeyFromKeyring`↗, `Key`↗ is called, that retrieves the key↗ by calling the `Get` method of a key stored in a file. The `Get`↗ method calls `unlock` that checks if the password was not already retrieved in the past↗, in which case it calls `passwordFunc`↗ which for a file-backend key corresponds to `newRealPrompt`↗. `newRealPrompt` checks if the password corresponds to the stored key hash↗ (hash in the `keyhash` file that contains the hash of the `passphrase`) and returns successfully if this is the case. If the `passphrase` is incorrect, the password check↗ fails, and hence the `unlock`↗ fails due to a wrong password, and as a result the `Unlock` fails as well↗.

From what described so far, note that if a user knows the `passphrase`, the user can successfully use that `passphrase` to load the key and then `getEOTSPrivKey`↗ (that calls `eotsPrivKeyFromKeyName`↗ that retrieves the key↗ from the `privateKeys map`) to use for signing (e.g., in `SignEOTS`↗).

Property EOTSD-02: The passphrase is never stored on disk (i.e., not in plain text). Similarly, the private key is never stored unencrypted on disk.

Conclusion

When we create a new key↗, `NewAccount`↗ is called that calls `writeLocalKey`↗ which calls `writeRecord`↗ that calls the underlying `Set`↗. `Set` for a file-backend key stores the private key encrypted↗. Similarly, only the `bcrypt` hash of the `passphrase` is ever stored on disk as can be seen here↗. Finally, the private key can only be decrypted through the correct use of the password↗.

Property EOTSD-03: The introduction of the mutex does not cause any race conditions, live- or dead-locks

Findings: EOTSD - A lock missing in getKeyFromMap

Conclusion

The `LocalEOTSManager`↗ contains the single `mu` mutex and hence a potential deadlock can stem if `mu` is `Locked` twice without being `Unlocked` first. Note that `mu` is used only in two places: (i) `Unlock`↗, and (ii) in `getEOTSPrivKey`↗. Both `Unlock` and `getEOTSPrivKey` correctly use `defer` and are correctly used to protect the `privateKeys`↗ `map`. The `getEOTSPrivKey` method is called by sign-related methods (`SignEOTS`↗, `UnsafeSignEOTS`↗, and `SignSchnorrSig`↗), as well as `KeyRecord`↗ and neither of those methods end up calling `Unlock`. Reversely, `Unlock` or its internal method calls do not call `getEOTSPrivKey`. Finally, note that `mu` is used to protect the access to `privateKeys`: `privateKeys` is written during `Unlock`↗ while the lock is being held and `privateKeys` is read during `getKeyFromMap`↗ and hence the `privateKeys map` is read in `eotsPrivKeyFromKeyName`↗. `eotsPrivKeyFromKeyName` is called by `getEOTSPrivKey`↗ that acquires the lock before reading `privateKeys` so everything is fine. The `eotsPrivKeyFromKeyName` method is also called by `SignSchnorrSigFromKeyname`↗ without holding the lock. Nevertheless, `SignSchnorrSigFromKeyname` is used for the `export` command that is not in use anymore (see relevant finding).

Property INT-01: Integration of new modules does not interfere with the chain functioning

Violation consequences

This property is about checking that the integration is done following the best practices and that newly added modules do not create any problem for the chain's normal functioning. For the most modules (except for the rate limiter), we want the new modules not to introduce any new behavior. Instead, they just set the ground for future features.

Threats

- There may be a technical mistake integrating a new module, such as:
  - mixing up the order of middlewares
  - not adding new fields to the store (similarly, not deleting obsolete fields)
  - using wrong parameters when initializing a new module
- A new module's behavior could harm the regular functioning of the chain (This may come from a technical mistake as listed above, but also from misalignment of the chain's characteristic's and a module's default use-case.)

Conclusion

Checking all the newly introduced module, we found that they are integrated correctly. In particular:

- The ordering of middlewares is correct.
- The token creation fee is set↗ to be `10_000_000 ubbn` = `10 bbn`. This makes DoS style attacks expensive.
- The capability `EnableCommunityPoolFeeFunding`↗ is enabled, which will send fees to the community pool upon token creation.
- Since the fee is set in `bbn`, creation of a token will only send a fee in `bbn` to the community pool. The fee is sent directly↗ (thus, no custom handling of the transfer fees).
- There is no looping over denoms anywhere in Begin/End blockers, and there is a cost of `10 bbn` assigned to creation of each denom, making it mildly costly to automate creation of large sequences of new tokens.
- The capability `enable_admin_sudo_mint` is enabled↗, while the function for determining whether an address is a sudo admin is set to↗ `DeafultIsAdminSudoFunc`↗, which always returns false. (Also noted in the ***Miscellaneous Comments*** finding.)
- There is a rate limit added↗ for each transfer channel for `ubbn` . The limit is on 20% of total supply per day
- Note: There are no whitelisted pairs (for exceptions / potential emergency transfers).
- Note: Tokenfactory-created tokens won't have rate limiters. (They are supposed to be added per governance once a new token is created).
- There are checks for the sizes of IBC messages↗, both for `Transfer` and `ICA` transactions.
- While `Host` is enabled (in the default params↗), the set of allowed messages is set to↗ `nil`. This makes for a safe start of the integration with no active ICA connection.
- The fact that Babylon's unbonding period is shorter (compared to a typical Cosmos chain) does not present a security issue for PFM: as long as relayers are able to relay IBC packets, all the guarantees stay in place.

Property RWD-01: For each height h, all finality providers that voted for the block at height h within a timeout TIMEOUT eventually allocate the correct reward

Threats

- The voting power table (distribution cache) that is used at some height *h'* to allocate the rewards for height *h* does not match the distribution table at the height *h*.
- The information which finality provider voted at height *h* is incorrect or from the wrong height.
- A reward is not allocated for a height *h* although Finalization is enabled and the finality provider has non-zero delegations and voted for *h*.

Conclusion

The property holds.

- The first threat is not possible. Each power update is recorded at the beginning of each block for that block height and those distributions are afterwards used for that specific height when the events are processed. There is no possibility of using a distribution from another height. The changes in the reviewed diff do not make this threat possible.
- The second threat is not possible. The stored FP votes for the block at a specific height have not been affected by this diff. These are gathered and used when active FO are being rewarded here.
- The third threat is not possible. If there is a FP that needs to be rewarded at height *h*, this cannot be missed because all the data structures used for rewarding within the store have been filled properly and latter used at appropriate height processing. The changes in the reviewed diff do not make this threat possible.

Property RWD-02: For each heights h1 and h2, if h1 < h2 the rewards for h1 must be allocated before the rewards for h2 are allocated

Threats

- `Keeper.RewardBTCStaking` (`x/incentive`) is called with a height *h* that is smaller than the last last height it was called.

Conclusion

This property holds.

It is assumed that finalization happens sequentially but there might be gaps in the sequence if there was no staked BTC at that height .

The allocation of rewards is done by calling `Keeper.rewardBTCStaking` by the following code:

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

Assume by contradiction that *h2* > *h1* and the rewards for *h2* are allocated before the rewards of *h1*. If `rewardBTC-Staking` is called for *h2*, `nextHeightToReward` is set to *h2* + 1. In further iterations of the loop rewards for heights higher or equal to *h2* + 1 are allocated. Thus the function never rewards *h1* after *h2*, a contradiction.

Property RWD-03: All rewards for height h must be allocated before a reward that includes height h is distributed

Threats

- A delegator changes its delegation at height *h*. The rewards for height *h-1* are added to `FinalityProviderCurrentRewards[i]` after the rewards for height *h-1* are distributed to the delegator.

Conclusion

The property holds.

When the event are being processed they are being stored to a reward tracker event. These events are later used when the reward are being distributed. The distribution always happens from the last processed height up to the height to examine (current height - `params.FinalitySigTimeout` - height from which the finality providers have had enough time to cast their finality votes). Rewarding is done height by height first by processing all the events for reward tracker until that block height in `ProcessRewardTrackerEvents`. It is called only for the unprocessed heights up to the desired height and processes the events in `ProcessRewardTrackerEventsAtHeight` that gets all the events for that block height, process those events updating the reward tracker structures and deletes all the events processed.

The mechanism of allocation and distribution did therefore not change with the diff introduced by the PR. In `btcDelegationModifiedWithPreInitDel`, first the current period ends and the rewards up to height *h-1* are distributed. Then the rewards for height *h* are allocated by applying the changes to `FinalityProviderCurrentRewards`.

Property RWD-04: For each height h, the rewards for a height h and a delegator d are distributed at most once

Threats

- Events are processed (`BTCDelegationActivated/BTCDelegationUnbonded`) for a different height as they did occur.
- There are two or more events for a delegator at height *h*, and the rewards until height *h* are distributed each time the event is processed.

Conclusion

The property holds.

- Event (`BTCDelegationActivated/BTCDelegationUnbonded`) processing is done in the `ProcessAllPowerDistUpdateEvents` function here↗ eventually through `MustProcessBtcDelegationActivated` and `MustProcessBtcDelegationUnbonded` functions. These use `AddEventBtcDelegationActivated` and `AddEventBtcDelegationUnbonded` respectively, and call these for the current context height. These events have been collected from the previous height but this is done consistently for all of them. No deviation is possible and this process has not been changed through the reviewed diff.
- Multiple distribution of rewards is not possible. The reviewed diff did not make this threat possible. Events are processed one by one in the `BeginBlock` and the rewards are afterward distributed at most once for each delegator if conditions are met.

Property RWD-05: Rewards for height h must be distributed according to the actual delegations at the height, and based on which FP actually voted for the block at height h until reward distribution

Threats

- Not all changes to the delegation are recorded when calculating the reward, or changes are recorded that did not happen.
- The delegation table was not properly initialized.
- Delegation changes are applied at the wrong height.
- The information which FP voted at height *h* that is used for the calculation does not reflect the voting for height *h*.

Note: We did not audit the actual reward calculation as this was not changed by the PR under audit and is therefore out of scope.

Conclusion

The property holds under premise that the implementation of the reward calculation was correct before.

- As in the protocol before the PR changes, rewards are distributed based on the `FinalityProviderHistoricalRewards` and the `BTCDelegationTracker`. There is a individual delegation tracker for each FP and delegator. Every time a delegation is activated/unbonded, the amount added/subtracted from the `TotalActiveSet`. The tracker is initialized at the beginning of a period (i.e. the last time the delegation changed) by `initializeBTCDelegation` by starting new period (set to `FinalityProviderCurrentRewards[fp]-1`) and the same delegation (`TotalActiveSat`) as in the previous period. This mechanism works the same as in the previous version of the protocol, if events are passed to the functions `BtcDelegationActivated` or `BtcDelegationUnbonded` for the right height and in the same order as they occurred.
- The tracking is implemented by maintaining the `rewardTrackerEvents` map. The map is initialized at genesis from `gs.EventRewardTracker`. An event is added in `BeginBlocker` if and only a change in the power distribution happened (`processRewardTracker` with `MustProcessBtcDelegationActivated` or `MustProcessBtcDelegationUnbonded`). When rewards are distributed until a height `untilBlkHeight`, in `ProcessRewardsTrackerEvents` events are processed for all heights `lastProcessedHeight + 1` to `untilBlkHeight`, this value is set at the end as `lastProcessedHeight`. The events are processed in the order they occurred at that height and the same functions as in the previous versions of the module were called.
- Delegation changes are tracked and processed for the correct height, i.e. the height when the delegation change occurred.
- The information which FP voted at height *h* that is used for the calculation does correctly reflect the voting for height *h*. The votings are taken from `k.GetVoters(_,h)` in `rewardBTCStaking` for height *h*. The total voting power is computed based on this and the `VotingPowerDistCache` which was taken correctly from heigh *h* in `rewardBTCStaking`. In addition the reward is allocated only if there is an entry in that map for the respective finality provider.

Property VIG-01: Exactly-once and height ordered Babylon blocks processing in BTC staking tracker

Findings: VIG - Enforce monotonic block height processing on Vigilante, VIG - Vigilante processes Babylon data using unsynchronized block snapshots tracked, VIG - Potential duplicate event processing due to overlapping block height ranges

For every Babylon block *b*, if *b* contains delegation-related events, the Vigilante must: detect and process *b* exactly once, and in height order (i.e., height *h*, then *h+1*, etc.).

Violation consequences

- Missing processed blocks could potentially lead to faulty activation or reporting of unbonding to Babylon chain.

Threats

- Staking tracker doesn't contain information about last processed Babylon block height or doesn't update it after processing is done.
- Babylon blocks processing of events and delegation statuses is overlapping due to concurrent execution.
- BTC staking tracker tracking will contain invalid on-chain delegation status for Babylon's last processed block b.

Conclusion

The property does not hold.

1. Vigilante contains information about last processed Babylon height block `currentCometTipHeight` (code ref↗). The value is updated upon:
   - the (re)start of the Vigilante - the value is set to the latest block height on Babylon chain (code ref↗),
   - each ticker the CometBFT block fetcher routine queries for current Babylon block height (code ref↗) and in case it differs (code ref↗) from `currentCometTipHeigh`, it fetches the missing block span events from Babylon chain (code ref↗):

   ```
   if err := sew.fetchDelegationsByEvents(sew.currentCometTipHeight.Load(),
   ↪  latestHeight); err != nil {
     return fmt.Errorf("error fetching delegations by events: %w", err)
   }
   ```

   and updates the last processed block, upon the operation is done (code ref↗). The assumption is that Vigilante is working with a progressing Babylon block chain.
   It was concluded that a check enforcing monotonic block height processing is currently missing on Vigilante. This informational issue has been reported accordingly.
2. Upon (re)start of the Vigilante, the `currentCometTipHeight` is set (code ref↗) and a bootstrapping process is triggered to align Vigilante with the latest Babylon blockchain state. All the existing delegations are retrieved with `checkBabylonDelegations` (code ref↗) as part of the bootstrapping mechanism. These delegations are retrieved with a query performed over the latest Babylon block height (code ref↗), which might differ from the stored `currentCometTipHeight` due to blockchain progressing from the moment of storing `current-CometTipHeight` in the Vigilante. This misalignment between tracked and queried Babylon blocks caused unexpected Vigilante executions. This issue has been reported accordingly.
3. Events are fetched per event type concurrently (code ref↗). Within a single ticker tick, we spawn concurrent execution of event fetches. However, it still awaits them all to finish (code ref↗) and the resulting `stakingTx-Hashes` are executed sequentially after removing duplicates. We concluded that there is no possibility for unordered blockheight processing due to this sequential processing per tick, and since the `stakingTxHash` is used to retrieve the `BTCDelegation` containing the latest on chain status. However the criteria defined for the `StakingTxHashesByEvent` (code ref↗) was detected to contain an issue when the criteria is propagated in `RPCClient.TxSearch`. This issue has been reported accordingly.
4. Further, the retrieval of BTC delegation statuses relies on the latest block state on Babylon, which may lead to inconsistencies if not aligned with the event processing height in Vigilante. This concern is further discussed in the ***Property VIG-05*** and is conceptually based on the issue described in the issue and is similar with the observation made in point 2). Since events are processed in order and delegation data is retrieved by `stakingTxHash`, there is no risk of Vigilante reverting to outdated delegation statuses after processing newer events.

Property VIG-02: No skipped blocks in BTC staking tracker processing

Vigilante must process all Babylon blocks from the last processed Babylon block height *h*, to the current Babylon block height, e.g. *h+n* with each processing Vigilante tick round.

Violation consequences

- Skipped blocks in processing could lead to missed delegation transitions, important for activation and unbonding of the delegation.

Threats

- Staking tracker doesn't contain information about last processed Babylon block height or doesn't update it after processing is done.
- During Vigilante (re)start, improper synchronization between delegation state initialization and block height tracking can cause gaps in processed Babylon blocks or missed delegation updates.
- During Vigilante (re)start, the daemon may fail to properly process all delegations, potentially resulting in tracking stale or inconsistent staking state.

Conclusion

The property holds.

As concluded in the ***Property VIG-01***, the Vigilante correctly tracks and updates last processed Babylon chain block in `currentCometTipHeigh`.

Upon (re)start, `currentCometTipHeight` is initialized to the latest Babylon block height (code ref↗), after which the bootstrapping process - executed by the `fetchDelegations` routine - is triggered. Once this initial delegation fetching completes, the periodic block processing begins via the `fetchCometBftBlockForever` routine, which operates on a fixed interval. This routine starts processing from the last recorded `currentCometTipHeight` up to the current Babylon block height - `latestHeight` (code ref↗).

During the bootstrapping phase of Vigilante, all existing delegations - regardless of their status - are queried from the Babylon chain starting from the latest block height (or even a greater height, as explained in the associated issue).

As discussed in the reported ***VIG: Potential duplicate event processing due to overlapping block height ranges*** issue, overlaps can occur due to the query criteria used when fetching events for the block span [`currentCometTipHeight`, `latestHeight`]. However, no blocks are skipped during this process.

An additional ***VIG: Bootstrapping and block fetching race condition risk*** issue was observed where delegation statuses fetched from newer blocks during periodic execution were overwritten by older statuses retrieved during bootstrapping from lower block heights. While this scenario is highly unlikely due to `fetchCometBftBlockForever` being triggered only on periodic ticks, it is not technically a missed block - but the consequence is functionally equivalent, as more recent data is lost.

Nevertheless, the Vigilante block tracking mechanism ensures that all blocks are processed, maintaining continuity in block coverage; therefore, this property holds. While the related issues discussed above may lead to similar consequences, they affect the correctness of other associated properties (***Property VIG-01***, ***Property VIG-04***).

Property VIG-03: Vigilante must process events only for Babylon blocks that were not previously processed

Findings: VIG - Enforce monotonic block height processing on Vigilante, VIG - Potential duplicate event processing due to overlapping block height ranges

If there are no new Babylon blocks since the last processed height, the BTC staking tracker must not perform any processing of Babylon events.

Violation consequences

- Redundant or duplicate processing might lead to unnecessary computation or reprocessing or potentially re-handling already processed events and delegation status transitions.

Threats

- Incorrect handling of last processed block and current Babylon block logic.
- Timer-based triggers run processing rounds even when there's no new data.

Conclusion

The property does not hold.

If `currentCometTipHeigh` (last processed block height on Vigilante) is equal to current Babylon block tip `latestHeight` (code ref↗) further processing of events and delegations querying is aborted.

As detailed in ***Property VIG-01*** and the related finding analysis, the criteria for fetching events from newly produced Babylon blocks are incorrectly defined, causing Vigilante to process events at boundary block heights multiple times.

Property VIG-04: Isolated and sequential block span processing

Findings: VIG - Bootstrapping and block fetching race condition risk

Vigilante must isolate processing across block height spans to prevent overlapping or concurrent execution. While it is processing blocks in the range from *h* to *h+n*, it **must not initiate processing** of any new block height spans that either overlap or begin beyond *h+n*. Additionally, **delegation accounting must not occur in parallel** with new event processing. This ensures that state updates - such as delegation transitions - are applied within a non-overlapping interval.

Violation consequences

- Failure to enforce strict isolation and sequential processing may result in:
  - Duplicate processing of the same events,
  - Race conditions in delegation state tracking,
  - Faulty activation or unbonding delegations due to inconsistent final delegation state.

Threats

- Improper concurrency control: Missing mutexes, race conditions over shared channels, or lack of atomic coordination between event and delegation processing logic.

Conclusion

The property does not hold.

Vigilante processes newly created Babylon blocks using the `fetchCometBftBlockForever` routine, which runs on ticks and sequentially processes a block span from the last processed Babylon block height `currentCometTipHeight`, tracked in the Vigilante, to the `latestHeight`, current latest height in Babylon chain.

For each tick, events are fetched concurrently by event type (code ref↗). Although these fetches are executed in parallel, the system waits for all of them to complete (code ref↗). Afterward, the resulting `stakingTxHashes` are deduplicated and processed sequentially. Therefore, the periodic block fetching routine will be executed sequentially.

The reported ***VIG: Potential duplicate event processing due to overlapping block height ranges*** issues related to incorrectly defined event query criteria and the mismatch between the latest Babylon block height, the event snapshot, and `currentCometTipHeight` (as discussed in ***Property VIG-01****) do* not impact the correctness of this property.

This property specifically addresses race conditions and execution order issues. While the query criteria may result in overlapping block processing, such overlaps are orthogonal to the concurrency risks covered here and do not compromise the correctness of this property.

Another aspect involves analyzing the potential parallel execution of the bootstrapping and periodic block-fetching routines (code ref↗).

The bootstrapping `fetchDelegations` routine queries all delegations present in the latest Babylon chain state (noting that `currentCometTipHeight` may not reflect the actual latest block height, as discussed in the linked issue). This data populates Vigilante's tracking structures for pending and unbonding delegations. Since bootstrapping may take several minutes while the chain continues producing blocks, it's critical to avoid missing events during this period.

The execution order of `fetchDelegations` and `fetchCometBftBlockForever` routines can affect the outcome, so it was analyzed to conclude if they could execute in different order, in parallel or only sequentially.

The `fetchDelegations` routine is designed to run first, initializing Vigilante state before periodic block processing (`fetchCometBftBlockForever`) begins. This typically occurs as `fetchCometBftBlockForever` operates on interval-based ticks (code ref↗). During bootstrapping, Babylon block fetching is paused using the `delegationRetrieval-InProgress` flag, which blocks `FetchCometBFTBlock` execution while delegation data is being retrieved (code ref↗). This atomic boolean guard is applied when the delegation query is submitted (code ref↗) and ensures that Babylon CometBFT blocks are fetched only after (code ref↗) the delegation data has been fully retrieved (code ref↗).

However the implementation does not contain the explicit synchronization of the two go routines.

An appropriate issue was reported.

Property VIG-05: Latest delegation status must be used when multiple events occur for the same delegation in a height span

Findings: VIG - Vigilante processes Babylon data using unsynchronized block snapshots tracked

If multiple events for the same delegation appear in a block height spans being processed *h, h+n* (e.g., covenant proof and unbonding), the tracked delegation must be processed only once with latest delegation status on Babylon.

Violation consequences

- Tracking of delegation in incorrect Vigilante tracking structure could lead to incorrect activation or unbonding delegations.

Threats

- Conflicting events for the same delegation *d* (referencing single staking tx hash) were retrieved in processed block span *h, h+n.* Analyze the resulting delegation state and determine whether any expected status transitions were missed or not tracked.

Conclusion

The property holds.

Vigilante uses events as a signal to fetch the actual delegation by `stakingTxHash`.
The Vigilante always performs queries over the latest Babylon chain state for the observed block span. Analysis concluded that events are not a source for determining the on-chain status of delegations, but the correct status is obtained by querying the node via an RPC call (code ref↗).

As detailed in ***Property VIG-01*** and the related finding analysis the unexpected Vigilante executions may arise from misalignment between tracked (`currentCometTipHeight`) and queried Babylon blocks (queried Babylon block height > `currentCometTipHeight`).

While the BTC staking tracker does not always query delegation status at the exact expected Babylon block height, this behavior does not lead to incorrect delegations processing. The BTC staking tracker includes safeguards for handling delegations that are with an unexpected status. Specifically, if a delegation appears in a later-than-expected status (e.g., from the latest +1 block height), this typically means that another Vigilante instance has already processed a relevant event and updated the state on the Babylon chain accordingly. While processing `pendingTracker` and `unbondingTracker` to activate or unbond delegations and if the Babylon node indicates that a delegation is not in the expected status on Vigilante, it will be removed from the tracking structure. Since the on-chain status reflects the most recent delegation state, Vigilante reacts appropriately by removing or adjusting the tracked delegation. As a result, no incorrect or outdated delegation states are acted upon.

Although delegation status may be fetched from a slightly later height than expected, the design of Vigilante ensures that only the latest valid on-chain state is acted upon. There is no negative consequence or incorrect behavior observed from this mechanism. The property is upheld in practice due to the Vigilante's robustness in handling real-time status changes.

Property VIG-06: Accurate delegation tracking for activation and unbonding

Findings: VIG - Suggestion to track UNBONDED and EXPIRED delegation statuses in Vigilante

Vigilante must accurately track all relevant events and delegation status transitions to ensure correct activation and unbonding of delegations on the Babylon chain.

Violation consequences

- Failure to capture the correct events or delegation status transitions may lead to incorrectly submitted unbonding or slashing transactions, resulting in unintended penalties or lost funds.

Threats

- Incomplete event or delegation status tracking in Vigilante.

Conclusion

The Vigilante BTC staking tracker monitors specific Babylon events to track delegation state transitions and trigger corresponding actions. The following event types are tracked as part of the implementation (code ref↗):

- `EventCovenantQuorumReached`
- `EventBTCDelegationInclusionProofReceived`
- `EventBTCDelegationStateUpdate`

These events are used to maintain internal tracking structures:

- **`unbondingTracking`**: Stores delegations in `VERIFIED` and `ACTIVE` statuses. On notification of a new BTC block, this structure is checked. If an `unbondingTx` is present, Vigilante submits a `MsgBTCUndelegate` and subsequently removes the delegation from tracking.
- **`pendingTracking`**: Stores delegations in the `VERIFIED` status. Upon new BTC block notifications, if a `stakingTx` is found, Vigilante submits a `MsgAddBTCDelegationInclusionProof` and removes the delegation from the structure.

While processing `pendingTracker` and `unbondingTracker` to activate or unbond delegations, if the Babylon node indicates that a delegation is not in the expected status, it will be removed from the tracking structure, since it is not possible to perform the intended action of delegation activation or unbonding. Therefore, the initial concern that Vigilante might keep delegations out of order in memory that could not be processed, does not materialize in practice.

The analysis concluded that Vigilante tracks a sufficient set of event types to cover all necessary delegation status transitions, enabling correct activation or unbonding of delegations on Babylon. This conclusion is based on: (1) modeling the expected delegation lifecycle and its associated events based on the `x/btcstaking` module's emission logic, (2) comparing this model against the actual tracking implementation in Vigilante, and (3) focusing how the Bitcoin-side events critical for triggering correct unbonding or activation messages are processed in Vigilante.

One informational issue proposing additional tracking in the BTC staking tracker has been reported; however, it requires careful consideration due to performance priorities.

Findings

Finding	Type	Severity	Status
VIG - Enforce monotonic block height processing on Vigilante	Implementation	Informational	Resolved
VIG - Vigilante processes Babylon data using unsynchronized block snapshots tracked	Implementation	Low	Acknowledged
VIG - Potential duplicate event processing due to overlapping block height ranges	Implementation	Medium	Acknowledged
VIG - Bootstrapping and block fetching race condition risk	Implementation	Medium	Resolved
VIG - Suggestion to track UNBONDED and EXPIRED delegation statuses in Vigilante	Protocol	Informational	Acknowledged
BAB - Missing validation for haltingHeight smaller than or equal to currentHeight in finality provider halting logic	Implementation	Medium	Resolved
BAB - Minor code improvements in x/finality module	Implementation	Informational	Resolved
BAB - ValidateBasic functions explicitly called in x/btcstaking msgServer	Implementation	Low	Resolved
GEN - Missing validation for epoch existence in SubmissionEntry during InitGenesis	Implementation	Informational	Resolved
GEN - Missing Minter validation in InitGenesis of x/mint module	Implementation	Medium	Resolved

Finding	Type	Severity	Status
GEN - x/btcstaking genesis state is not entirely validated with ValidateGenesis	Implementation	Medium	Resolved
FP - A lock missing in getKeyFromMap	Implementation	Low	Resolved
FP - The TestEotsdUnlockCmd does not test unlocking properly	Implementation	Low	Resolved
Miscellaneous Comments	Implementation	Informational	Resolved

VIG - Enforce monotonic block height processing on Vigilante

Severity Informational **Impact** 1 - Low **Exploitability** 0 - None
Type Implementation **Status** Resolved

Involved artifacts

- `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go`↗

Description

There is a missing check enforcing strictly increasing block height processing in Vigilante (code ref↗).

Problem Scenarios

Currently, the only condition is (code ref↗) that `currentCometTipHeight` differs from latest Babylon chain block height retrieved (code ref↗):

```
latestHeight, err := sew.babylonNodeAdapter.CometBFTTipHeight(ctx)
...
if latestHeight == sew.currentCometTipHeight.Load() {
  sew.logger.Debugf("no new comet bft blocks, current height: %d",
↪  sew.currentCometTipHeight.Load())
  return nil
}

if err := sew.fetchDelegationsByEvents(sew.currentCometTipHeight.Load(),
↪  latestHeight); err != nil {
  return fmt.Errorf("error fetching delegations by events: %w", err)
}
```

While this check may not be strictly necessary - since the `TxSearch` criteria inherently prevent conflicting criteria and results (code ref↗) and would return an empty set - it nevertheless ensures the expected, increasing block height execution in Vigilante.

Recommendation

Add enforcement of monotonic block height processing in Vigilante. This check would ensure that blocks are processed in strictly increasing order, which is essential for maintaining accurate alignment with the current on-chain delegation state. If a block with a non-increasing height is detected, further processing would be aborted.

Even though this is highly unlikely to happen, the check would guarantee correct processing.

Status

Resolved↗.

VIG - Vigilante processes Babylon data using unsynchronized block snapshots tracked

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `vigilante/btcstaking-tracker/stakingeventwatcher/expected_babylon_client.go`↗
- `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go`↗

Description

Upon (re)start of the Vigilante, the `currentCometTipHeight` is set (code ref↗) and a bootstrapping process is triggered to align Vigilante with the latest Babylon blockchain state. All the existing delegations are retrieved with `checkBabylonDelegations` (code ref↗) as part of the bootstrapping mechanism. These delegations are retrieved with a query performed over the latest Babylon block height (code ref↗).

Once the bootstrapping mechanism completes, the `fetchCometBftBlockForever` routine may begin triggering on each tick, fetching events from newly created Babylon blocks.

Problem Scenarios

Potential issue here is that `currentCometTipHeight` could be set to `CometBFTTipHeight` (code ref↗) at the moment of querying this information. Meanwhile, the blockchain continues to progress. As a result, `fetchDelegations` may query new, untracked block heights in Vigilante that are ahead of the stored `currentCometTipHeight`. These newly queried heights will not be stored as the updated `currentCometTipHeight` but represent the latest state at the time of fetching delegations from the Babylon chain.

Since CometBFT fetching begins only after the bootstrapping completes, it processes blocks starting from the stored `currentCometTipHeight` up to the latest Babylon block height (code ref↗). However, the bootstrapping process may already have processed blocks beyond the height recorded in `currentCometTipHeight`. As a result, when `fetchCometBftBlockOnce` runs from `currentCometTipHeight`, some blocks may be processed more than once.

Recommendation

Delegations should be queried precisely at the `currentCometTipHeight` to ensure synchronization between Vigilante and the Babylon snapshot tracking. This approach effectively prevents unnecessary duplicate processing and helps avoid potential, though not observed in this analysis, issues with delegation activation and unbonding management.

The Babylon team has acknowledged this finding and opened the issue Staking event watcher bootstrap at block height↗ with plans to address it in a future release.

VIG - Potential duplicate event processing due to overlapping block height ranges

Severity Medium **Impact** 1 - Low **Exploitability** 3 - High

Type Implementation **Status** Acknowledged

Involved artifacts

- `vigilante/btcstaking-tracker/stakingeventwatcher/expected_babylon_client.go`↗
- `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go`↗

Description

`fetchDelegationsByEvents` calls `fetchStakingTxsByEvent` to fetch staking transactions from the Babylon chain. Events are fetched with the criteria defined for the (code ref↗) Comet BFT's `RPCClient.TxSearch` query:

```
res, err := bca.babylonClient.RPCClient.TxSearch(ctx, criteria, false, page,
 ↪  count, "asc")
```

Problem Scenarios

The `StakingTxHashesByEvent` (code ref↗) defines the criteria for the `RPCClient.TxSearch` query performed for each event type as (code ref↗):

```
criteria := fmt.Sprintf(`tx.height>=%d AND tx.height<=%d AND %s.new_state
 ↪  EXISTS`, startHeight, endHeight, event)
```

This defines an inclusive block height range `[startHeight, endHeight]`, where:

- `startHeight` is loaded from `sew.currentCometTipHeight`, and
- `endHeight` is the latest block height.

After fetching, `currentCometTipHeight` is updated to `latestHeight`.

Because the query uses an inclusive range (`tx.height >= startHeight AND tx.height <= endHeight`), and `startHeight` is set to the stored `currentCometTipHeight` (which has already been processed in a previous call), there is a risk that some blocks - particularly those at the boundary heights - may be processed more than once.

This can result in duplicated event processing and unnecessary overhead.

Recommendation

Adjust the block height range criteria to use exclusive lower bounds for subsequent queries, e.g., `tx.height > previousHeight` instead of `tx.height >= previousHeight`, to avoid overlapping block inclusion.

The Babylon team has acknowledged the issue and plans to address it in a future release.

VIG - Bootstrapping and block fetching race condition risk

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go`↗

Description

On startup, Vigilante must guarantee that the initial delegation state is fully initialized (via `fetchDelegations`) before any periodic block fetching (via `fetchCometBftBlockForever`) is allowed to process blocks or update tracking state. Overlapped blocks processing could lead to gaps or inconsistencies in tracked delegations, important for activation and unbonding of the delegation.

Problem Scenarios

The problem occurs if `fetchCometBftBlockForever` routine somehow runs first.

What could happen?

The atomic flag `delegationRetrievalInProgress` is set to `false`, allowing the periodic block fetching routine `fetchCometBftBlockForever` to execute. Depending on the pace of Babylon block production, the following outcomes are possible:

- If the latest Babylon block height is still equal to the Vigilante's `currentCometTipHeight`, the tick-triggered execution of `fetchCometBftBlockForever` will be skipped. Then, `fetchDelegations` will run, and Vigilante will complete its bootstrapping. This is the expected and correct execution order.
- If the Babylon chain has advanced such that `latestHeight > currentCometTipHeight`, then the tick-triggered `fetchCometBftBlockForever` may proceed before bootstrapping begins. In this case, the call to `fetchCometBft-BlockOnce` (code ref↗) results in block processing, including calls to `fetchDelegationsByEvents`. If this occurs before `delegationRetrievalInProgress` is set to `true`, the two routines could execute concurrently.

The consequence of this rare interleaving is that delegation statuses fetched from newer blocks by the periodic routine may be overwritten by older delegation states retrieved during bootstrapping from lower block heights. While no blocks are technically skipped, the end result may appear as if delegation data from some blocks was lost - effectively resembling a missed block scenario.

While theoretically possible, the risk of a race condition where `fetchCometBftBlockForever` begins execution before the `fetchDelegations` bootstrapping routine is extremely low in practice. This is because `fetchDelegations` is invoked immediately at startup, while `fetchCometBftBlockForever` waits for a 2-second ticker before processing - so it usually "loses the race". The only way it"wins" and executes first is if the scheduler delays `fetchDelegations` for over 2 seconds, which is unlikely - but not impossible. Under normal conditions - with few goroutines, no blocking at startup, and sufficient system resources - the Go scheduler is expected to start `fetchDelegations` first. However, rare delays in scheduling (e.g., preemption or CPU contention) could allow `fetchCometBftBlockForever` to proceed first, potentially leading to premature use of uninitialized BTC staking tracker in memory state.

This makes it a low-probability but noteworthy concurrency risk. However, the implementation could be improved to guarantee this race condition never occurs, ensuring deterministic behavior.

Recommendation

To guarantee correct execution order and eliminate the risk entirely, use synchronization like sync.Once, a signal channel, or a WaitGroup to ensure state initialization completes before periodic execution begins. An `atomic.Bool` flag could be initialized to `false` at Vigilante startup and only set to `true` after the bootstrapping (`fetchDelegations`) process completes. The `fetchCometBftBlockForever` routine should check this flag and delay execution until it is set, ensuring that periodic block fetching does not begin before delegation state is fully initialized.

Status

Resolved↗.

VIG - Suggestion to track UNBONDED and EXPIRED delegation statuses in Vigilante

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Protocol **Status** Acknowledged

Involved artifacts

- `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go`↗

Description

Currently, the Vigilante BTC staking tracker monitors delegation statuses `VERIFIED` and `ACTIVE` through specific Babylon-emitted events to manage delegation lifecycle transitions and trigger actions based on Bitcoin transactions and events.

However, it does not explicitly track the delegation statuses `UNBONDED` and `EXPIRED`, nor the accompanying events emitted by the Babylon chain: `EventBTCDelegationStateUpdate` with `UNBONDED` status and `EventBTCDelegationExpired` with `EXPIRED` status.

These statuses are not critical for delegation activation or unbonding submissions on the Babylon chain.

Problem Scenarios

Tracking of `UNBONDED` and `EXPIRED` delegation statuses presents an opportunity to enhance the completeness of BTC staking tracker's internal tracking structures (`pendingTracking`, `unbondingTracking`) to more accurately reflect the on-chain delegation status.

Incorporating explicit tracking of these statuses could improve Vigilante's ability to promptly and directly remove delegations that have completed unbonding or expired on-chain, further reducing the risk of stale or inconsistent state.

Recommendation

Enhance Vigilante's event tracking by explicitly subscribing to Babylon chain event: `EventBTCDelegationExpired`and process two additional delegation statuses `UNBONDED` and `EXPIRED`.

Leveraging these delegation statuses to proactively remove delegations from tracking structures upon unbonding or expiration will improve Vigilante's internal state accuracy and consistency. This enhancement will also strengthen monitoring to detect anomalies or invalid delegation states both on-chain and off-chain.

Since Vigilante already removes delegations when reporting unbonded or expired states, this event-driven approach serves as a proactive complement to existing cleanup logic, enhancing overall robustness.

Note: Potential performance impacts should be carefully considered to ensure this improvement does not reduce Vigilante's efficiency.

The Babylon team has acknowledged this as an enhancement (see issue Handle expired and unbonded event↗), and plans to implement it in the future.

BAB - Missing validation for haltingHeight smaller than or equal to currentHeight in finality provider halting logic

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `x/finality/keeper`↗

Description

The `HandleFinalityProviderHaltingProposal` function in the `x/finality` module assumes that the provided `haltingHeight` is not in the future but does not explicitly enforce this condition. As a result, when `haltingHeight > currentHeight`, certain logic is silently skipped (gov.go#L104↗), leading to inconsistent state transitions. Specifically, while finality providers are marked as jailed and their signing info is reset, their voting power remains unchanged in the historical cache, which can cause them to appear active despite being inactive.

Although proposals with such misconfiguration are unlikely to be approved in practice, the lack of defensive validation introduces risk.

Problem Scenarios

- If `haltingHeight` is in the future, the loop that clears voting power and updates the historical cache will be skipped due to an invalid range (`haltingHeight > currentHeight`).
- As a result, finality providers may be jailed and have their signing info reset (gov.go#L74-L94↗), but their voting power in the cache remains non-zero, causing them to appear active despite being inactive.
- Additionally, when `haltingHeight > currentHeight`, the `GetVoters` method will return `nil` (gov.go#L26↗). In Go, reading from a `nil` map (gov.go#L42-L46↗) returns false for all keys, making it seem as if no provider voted. All targeted finality providers will be penalized for missing the vote — even though the vote hasn't happened yet.

Recommendation

Add a validation step to explicitly check that `haltingHeight <= currentHeight` at the beginning of the handler. If this condition is not met, the proposal should be rejected or a clear error should be returned.

Status

Resolved↗.

BAB - Minor code improvements in x/finality module

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Resolved

Involved artifacts

- `x/finality/types`↗
- `x/finality/keeper`↗

Description

Several minor improvements have been identified in the `x/finality` module that can enhance code clarity, efficiency, and input validation, particularly within governance proposal handling and voting power calculations.

Problem Scenarios

- The check `if uint64(haltingHeight) < params.FinalityActivationHeight` is located after a call to `k.GetVoters(...)`, causing an unnecessary store read when the input is already invalid (gov.go#L28↗). Move the finality activation height check to precede `GetVoters` to avoid unnecessary reads.
- The list `fpPksHex` may contain duplicate public keys (gov.go#L35↗). Although the current use of `map[string]struct{}` avoids functional issues, it is preferable to reject duplicates explicitly using `ValidateBasic` to reduce redundant logging and maintain clean governance inputs.
- The total voting power (`TotalVotingPower`) is calculated in a second loop (power_table.go#L100-L103↗) even though it could be accumulated during the first iteration over `FinalityProviders`.
- `IsSlashed is true` should also be included in the function doc comment (power_table.go#L226-L227↗) as a condition when voting power is treated as 0.
- The `signingInfo.JailedUntil` field is set using `time.Unix(0, 0)` without explicitly normalizing to UTC (power_dist_change.go#L136↗). While this does not affect logic (since the zero time is stored identically in all time zones), printing or logging such values could lead to misleading timestamps depending on the local timezone of the node.

Recommendation

As explained in the section above.

Status

Resolved↗.

BAB - ValidateBasic functions explicitly called in x/btcstaking msgServer

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low
Type Implementation **Status** Resolved

Involved artifacts

- `x/btcstaking/keeper/msg_server.go`↗

Description

Due to changes introduced in Cosmos SDK v0.50, `ValidateBasic` is officially deprecated, as noted in the Cosmos SDK documentation↗.

However, the function remains part of the `sdk.Msg` interface and, if implemented, it will still be automatically invoked during the `CheckTx` and `DeliverTx` phases.

Since the Babylon chain is using Cosmos SDK v0.50.12 and still implements `ValidateBasic` for its messages, there is no need to explicitly call it within the `msgServer` handlers.

Problem Scenarios

`ValidateBasic` will be executed twice per `x/btcstaking` messages:

- `MsgBTCUndelegate` (code ref↗)
- `MsgCreateFinalityProvider` (code ref↗)
- `MsgEditFinalityProvider` (code ref↗)
- `MsgAddCovenantSigs` (code ref↗)

since the function is explicitly called in the begging of every msgServer handler processing the above listed messages.

Recommendation

Remove explicit `ValidateBasic` calls.

There may be additional occurrences of explicit calls elsewhere in the Babylon codebase; however, this was not analyzed in detail, as the full codebase was outside the scope of this audit. This issue was identified during the review of the genesis-related PRs.

Status

Resolved↗.

GEN - Missing validation for epoch existence in SubmissionEntry during Init-Genesis

Severity Informational **Impact** 0 - None **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `x/btccheckpoint/types/types.go`↗

Description

In the `x/btccheckpoint` module, the `InitGenesis` function does not validate whether the `Epoch` referenced in each `SubmissionData` actually exists in the `Epochs` state (types.go#L108-L120↗). Specifically, there is no check to ensure that `SubmissionData.Epoch` corresponds to an existing entry in the `EpochData` store (i.e., `EpochDataPrefix`). This creates a potential inconsistency in the genesis state, where a submission may point to a non-existent epoch.

Problem Scenarios

Although the `Epoch` field from `SubmissionData` is not currently used in functional code, future code changes that utilize this field could lead to runtime errors or unexpected behavior if invalid references are present in the genesis state.

Recommendation

Add a check during `InitGenesis` to ensure that for each `SubmissionData`, the referenced `Epoch` exists in the `Epochs` state. If the epoch is missing, initialization should fail with a clear error message indicating the problematic submission.

Status

Resolved (PR#1022↗, PR#1136↗).

GEN - Missing Minter validation in InitGenesis of x/mint module

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `x/mint/keeper/genesis.go`↗

Description

In the `InitGenesis` function of the `x/mint` module, the `Minter` object is not validated when it is non-nil (see genesis.go#L14-L16↗). While there exists a `Validate()` method on `GenesisState` that includes a call to `Minter.Validate()`, this method is not invoked during `InitGenesis`. As a result, invalid fields inside the `Minter` object (such as negative `InflationRate`, malformed `BondDenom`, etc.) could silently be accepted and stored in state without triggering any errors.

Problem Scenarios

Without proper validation of the `Minter` object during `InitGenesis`, invalid or inconsistent data may be written to state, potentially leading to incorrect inflation calculations or unexpected runtime behavior.

Recommendation

Call `gs.Validate()` at the beginning of `InitGenesis` to ensure that the `Minter` object is valid.

Status

Resolved↗.

GEN - x/btcstaking genesis state is not entirely validated with ValidateGenesis

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low
Type Implementation **Status** Resolved

Involved artifacts

- `x/btcstaking/types/genesis.go`↗

Description

Inconsistent validation logic could lead to invalid data being imported from malformed or maliciously updated export genesis file.

Problem Scenarios

After analysis of existing validations, we concluded that `x/btcstaking` genesis state is not entirely validated. Based on the comment found in the `Validate` function (code ref↗), it appears the Babylon team is aware of this. However, since the export functionality is currently unused, the necessary validations have not yet been implemented - even though new genesis state fields have been added (see code reference). (code ref↗):

- `allowed_staking_tx_hashes`
- `largest_btc_reorg`

The following genesis state fields are not validated at all:

- Finality provider fields should be validated with same checks as when creating or editing FP. (code ref1↗, ref2↗). Also, when BTC delegations are created it is validated that FP is not slashed (code ref↗).
- BTC delegation fields validations should be aligned with the existing validations in sdk.Msg `ValidateBasic` and other validations present:
  - `UnbondingTime` and `StakingTime` should be checked for valid values - code ref↗
  - re-checking if fields are empty or if there are stateless validations performed during ValidateBasic - code ref↗
  - `FpBtcPkList` should be validated for duplicate finality providers PKs - (code ref↗)
- BTC delegators are updated upon adding BTC delegations (code ref↗) so the `BtcPk` and `fpBTCPK` (code ref↗) must be validated.
- `EventIndex`:
  - index should be increased with each new event added in the `addPowerDistUpdateEvent` - so the exported genesis file should not contain events with the same indexes.
  - type of event should be appropriate for the delegation status - this should be verified upon initialization.

Recommendation

Validations currently implemented in the message service and in the `sdk.Msg`'s `ValidateBasic` functions should also be applied in the `x/btcstaking` module's `ValidateGenesis` logic.

Status

Resolved↗.

FP - A lock missing in getKeyFromMap

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low
Type Implementation **Status** Resolved

Involved artifacts

- `finality-provider/eotsmanager/localmanager.go`↗
- `finality-provider/eotsmanager/cmd/eotsd/daemon/pop.go`↗

Description

In the function `getKeyFromMap`, there is a read of the `lm.privateKeys` map. This read is not protected by a lock.

The comment above the read says that "*we don't call the lock here because we are already in the lock in caller function*". This is true for calling the function from (with some intermediate calls) `SignEOTS`. It is not true, however, when (with some intermediate calls) calling it from `exportPop`↗.

Problem scenarios

As the development team shared, the `exportPop` was only used historically, in the context of the airdrop. In that regard, it won't be a problem for regular chain functioning. However, the situation presents a high potential for future problems:

- using `exportPop` in the future, simultaneously with regular operations
- relying on the misleading comment in the future development

Recommendation

It would make sense to move the lock inside the `getKeyFromMap` function. Alternatively, if the `exportPop` function is not needed (as it was used only for the airdrop), probably it's the best to remove it entirely.

Status

Resolved↗.

FP - The TestEotsdUnlockCmd does not test unlocking properly

Severity Low **Impact** 3 - Low **Exploitability** 3 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `finality-provider/itest/e2e_test.go`↗

Description

The `TestEotsdUnlockCmd`↗ is not technically checking that the right `passphrase` unlocks the private key due to the `k.Password` being set↗ during the creation of the key↗. Because of this, any arbitrary `passphrase` can be used to unlock↗ the key.

Problem scenarios

This test creates a false sense of confidence and could be misleading for future maintainers or users of the code.

Recommendation

The test should be re-factored to include a happy path (correct `passphrase`) where the test succeeds, as well as a non-happy path (bogus `passphrase`) where the test fails.

Status

Resolved↗.

Miscellaneous Comments

Severity Informational **Impact** 1 - Low **Exploitability** 0 - None
Type Implementation **Status** Resolved

This finding lists points that do not present a security concern, but the audit team deemed worth sharing.

1. The capability `enable_admin_sudo_mint` is enabled↗, while the function for determining whether an address is a sudo admin is set to↗ `DeafultIsAdminSudoFunc`↗, which always returns false.

Comments in regards to the "Unlocking EOTSD keyring":

1. Typo: Instead of writing `[...] EOTS private key [...]`, it is written `[...] EOTX private key [...]` in unlock.go↗.
2. Comment `[...] or passPhrase is incorrect [...]` that appears in `SignEOTS`↗ and `SignSchnorrSig`↗ is slightly off in the sense that those methods do not take `passphrase` as a parameter. Those methods fail if the key failed to unlock due to an incorrect `passphrase` that later on leads the signing methods to fail.
3. This error message in `start.go`↗ needs to be updated to capture the fact that the keyring backend can also be `file` now.
4. Not clear if the documentation is to-be updated in some other PR (outside the audit), but current docs↗ need to be updated to encompass the recent changes. For example, the aforementioned doc writes (emphasis our own): "*The EOTS manager uses Cosmos SDK↗ backends for key storage. Since this key is accessed by an automated daemon process, it must be stored* ***unencrypted*** *on disk and associated with the test keyring backend.*" which is not the case anymore.
5. We suggest updating the documentation with clear instructions on how the move from a test- to a file-backend keyring is to take place and how the old test-backend keys are to be removed after the move. Although the upgrade instructions were outside the scope of the audited PR, properties such as "*[...] the private key is never stored unencrypted on disk*" are based on the fact that private keys stored previously using the test-backend are now removed.
6. A comment could be introduced here↗ to explain why we need to pass the `passphrase` twice, namely, because in case of a key being created for the first time, the key hash is not stored↗, and therefore the `passphrase` is read twice (here↗ and here↗).
7. The `defaultKeyringBackend`↗ should be updated as well to correspond to `keyring.BackendFile`.
8. The `TestEotsdUnlockCmd`↗ is not technically checking that the right `passphrase` unlocks the private key due to the `k.Password` being set↗ during the creation of the key↗. Because of this, any arbitrary `passphrase` can be used to unlock↗. The test can be re-factored to include a happy path (correct `passphrase`) where the test succeeds, as well as a non-happy path (bogus `passphrase`) where the test fails.

Status

Resolved (PR#1043↗, PR#483↗).

Appendix: Vulnerability classification

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

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

Impact Score

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

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)

ImpactScore	Examples
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

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

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

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

Severity Score

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

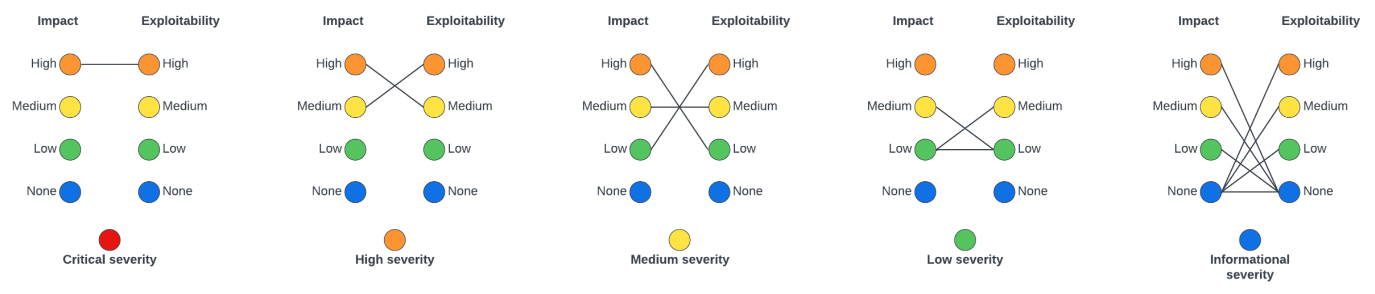

Figure 1: Severity classification

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

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

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.