// Security Assessment

09.30.2025 - 10.27.2025

Finality Provider + Covenent + Vigilante

Babylon Labs

HALBORN

Finality Provider + Covenent + Vigilante - Babylon Labs

Prepared by: HALBORN

Last Updated 01/22/2026

Date of Engagement: September 30th, 2025 - October 27th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
11	0	0	2	1	8

TABLE OF CONTENTS

7.11 Unused configuration variable: numpubrandmax

1. Introduction

Babylon Labs engaged Halborn to perform a security assessment of their core chain and peripheral services from September 30th, 2025 to October 27th, 2025. The assessment scope was limited to the repositories provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

2. Assessment Summary

Halborn was allocated 4 weeks for this engagement and assigned 1 full-time security engineer to conduct a comprehensive review of the assets within scope. The engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the peripheral services.
- Verify that the functionalities operates as intended.

In summary, Halborn identified several areas for improvement to reduce the likelihood and impact of security risks. All issues were resolved by the Babylon Labs team, except for informational severity findings that were acknowledged as intended design choice. The primary recommendations were:

- Implement garbage collection and rate limiting for VERIFIED delegations in Babylon to prevent zombie delegation accumulation.
- Modify Vigilante startup to query only recent VERIFIED delegations instead of ALL delegations.
- Prioritize unbonding detection over delegation activation to preserve economic security.
- Add duplicate height validation in EOTS Manager and Finality Provider daemon to prevent EOTS key extraction.
- Disable UnsafeSignEOTS endpoint in production via configuration flag or build tags.
- Add mutex locks to SignEOTS and SignBatchEOTS functions to prevent TOCTOU race conditions.

3. Test Approach And Methodology

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

The following phases and associated tools were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Goland variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts and unit tests.
- Static security analysis..

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORIES

(a) Repository: vigilante

(b) Assessed Commit ID: 67f06d8

(c) Items in scope:

- vigilante

Out-of-Scope: Third party dependencies and economic attacks.

(a) Repository: covenant-emulator

(b) Assessed Commit ID: ee05815

(c) Items in scope:

- covenant-emulator

Out-of-Scope: Third party dependencies and economic attacks.

(a) Repository: finality-provider

(b) Assessed Commit ID: 840dff5

(c) Items in scope:

- finality-provider

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- https://github.com/babylonlabs-io/finality-provider/pull/738
- https://github.com/babylonlabs-io/finality-provider/pull/736
- https://github.com/babylonlabs-io/finality-provider/pull/726
- https://github.com/babylonlabs-io/babylon/pull/1855
- https://github.com/babylonlabs-io/vigilante
- https://github.com/babylonlabs-io/vigilante/pull/482, https://github.com/babylonlabs-io/finality-provider/pull/737
- https://github.com/babylonlabs-io/covenant-emulator/pull/165
- https://github.com/babylonlabs-io/finality-provider/pull/735

- https://github.com/babylonlabs-io/finality-provider/pull/734

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	1	8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
COMPROMISED BABYLON NODE CAN EXTRACT EOTS PRIVATE KEY	MEDIUM	SOLVED - 11/05/2025
UNSAFESIGNEOTS EXPOSED IN PRODUCTION WITHOUT DISABLE OPTION	MEDIUM	SOLVED - 11/05/2025
TOCTOU RACE CONDITION IN SIGNEOTS AND SIGNBATCHEOTS	LOW	SOLVED - 11/05/2025
ZOMBIE VERIFIED DELEGATIONS BREAK ECONOMIC SECURITY VIA VIGILANTE/BITCOIN RPC DOS	INFORMATIONAL	ACKNOWLEDGED - 11/06/2025
VALIDATION GAP: FUNDING OUTPUT VALUE	INFORMATIONAL	SOLVED - 11/06/2025
STALE PARAMETER AFTER GOVERNANCE UPDATE	INFORMATIONAL	ACKNOWLEDGED - 10/28/2025
UNUSED CONTEXT PARAMETER PREVENTS CANCELLATION	INFORMATIONAL	FUTURE RELEASE - 11/06/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCONSISTENT ERROR LOGGING CONTEXT	INFORMATIONAL	SOLVED - 11/06/2025
INCONSISTENT VALIDATION ERROR CONTEXT	INFORMATIONAL	SOLVED - 11/11/2025
INCORRECT METRIC TYPES (GAUGE VS COUNTER)	INFORMATIONAL	SOLVED - 11/05/2025
UNUSED CONFIGURATION VARIABLE: NUMPUBRANDMAX	INFORMATIONAL	SOLVED - 11/05/2025

7. FINDINGS & TECH DETAILS

7.1 COMPROMISED BABYLON NODE CAN EXTRACT EOTS PRIVATE KEY

// MEDIUM

Description

A compromised or malicious Babylon RPC node can return duplicate block heights with different hashes, causing the finality provider to unknowingly sign the same height twice. Due to EOTS cryptographic properties, this mathematically reveals the EOTS private key through simple algebraic extraction.

Impact:

- **EOTS key extraction**: Signing same height twice reveals private key through simple algebra: `privKey = (sig1 - sig2) / (hash(msg1) - hash(msg2))`.
- **Trust model failure**: Documentation states running own Babylon node is "strongly recommended" but "not mandatory" - many FP operators will use third-party/public RPCs for convenience.
- **Attack vector**: Compromised or malicious Babylon RPC returns duplicate heights with different hashes → FP signs both → attacker extracts private key.
- **Complete validator compromise**: Attacker can submit both signatures to slash validator, confiscate BTC stake.
- **Undetectable attack**: Appears as normal RPC traffic, only detected after validator already slashed (irreversible damage).
- **Systemic risk**: Single compromised public RPC can extract keys from ALL connected validators simultaneously.
- **Security depends on external party**: FP security relies on Babylon RPC operator's integrity - violates defense-in-depth principles.

Code Location

In `finality-provider/eotsmanager/localmanager.go`, the EOTS Manager:

Copy Code

```
294 func (lm *LocalEOTSManager) SignBatchEOTS(req *SignBatchEOTSRequest) ([]SignDataResponse, error) {
295     // ❌ NO DUPLICATE HEIGHT VALIDATION
296
297     for _, request := range req.SignRequest {
298         height := request.Height
299
300         // Database check only prevents re-signing past heights
301         if existingRecord, found := existingRecords[height]; found {
302             continue  // Skip already signed
303         }
304
305         // ❌ No check if height appeared earlier IN THIS BATCH
306         // Same height signed twice with different messages → key extraction
307         privRand, _ := lm.getRandomnessPair(eotsPk, chainID, height)
308         sig := eots.Sign(privKey, privRand, request.Msg)
309         response = append(response, SignDataResponse{Signature: sig, Height: height})
```

```
    }
}
```

In `finality-provider/clientcontroller/babylon/consumer.go`, the FP Babylon RPC Consumer:

Copy Code

```go
func (bc *BabylonConsumerController) queryLatestBlocks(...) ([]types.BlockDescription, error) {
    res, err := bc.bbnClient.ListBlocks(status, pagination)

    // ❌ NO VALIDATION - trusts Babylon RPC response
    for _, b := range res.Blocks {
        blocks = append(blocks, types.NewBlockInfo(b.Height, b.AppHash, b.Finalized))
        // No duplicate height validation
    }
    return blocks, nil
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:C/C:C/A:C/I:C/D:C/Y:C (5.0)

Recommendation

- **EOTS Manager**: Validate no duplicate heights within batch before signing (defense-in-depth).
- **FP Daemon**: Validate Babylon RPC responses for duplicate heights before processing.
- Both layers should implement duplicate detection as security check.

Remediation Comment

SOLVED: The **Babylon Labs team** solved the issue in the specified commit.

Remediation Hash

https://github.com/babylonlabs-io/finality-provider/pull/738

7.2 UNSAFESIGNEOTS EXPOSED IN PRODUCTION WITHOUT DISABLE OPTION

// MEDIUM

Description

The `UnsafeSignEOTS` RPC endpoint is exposed in production and bypasses double-sign protection. If the FP daemon is compromised, an attacker can call this endpoint to sign the same height twice and extract the EOTS private key. No configuration option exists to disable this test-only endpoint.

Impact:

- **Bypasses slashing protection**: No database check means same height can be signed multiple times.
- **Key extraction if FP compromised**: Attacker with FP daemon access can call `UnsafeSignEOTS` twice with same height → extract EOTS private key.
- **Always enabled**: No configuration to disable in production - test endpoint permanently exposed.
- **Requires FP daemon compromise**: Mitigated by EOTS Manager being localhost-only, but if FP daemon is compromised, attacker has direct access.
- **Lower severity than FP_HIGH_1**: Requires compromising FP daemon (harder, needs host access) vs just malicious Babylon RPC (easier, network-based).

Code Location

In `finality-provider/eotsmanager/service/rpcserver.go`, the RPC server:

Copy Code

```
// UnsafeSignEOTS only used for testing purposes. Doesn't offer slashing protection!
func (r *rpcServer) UnsafeSignEOTS(_ context.Context, req *proto.SignEOTSRequest) (
    *proto.SignEOTSResponse, error) {
    sig, err := r.em.UnsafeSignEOTS(req.Uid, req.ChainId, req.Msg, req.Height)
    // ❌ No check if this is test mode
    // ❌ No way to disable this endpoint in production
    return &proto.SignEOTSResponse{Sig: sigBytes[:]}, nil
}
```

In `finality-provider/eotsmanager/localmanager.go` , the implementation:

Copy Code

```
// UnsafeSignEOTS should only be used in e2e test to demonstrate double sign
func (lm *LocalEOTSManager) UnsafeSignEOTS(fpPk []byte, chainID []byte, msg []byte, height uint64) {
    privRand, _ := lm.getRandomnessPair(fpPk, chainID, height)
    privKey, _ := lm.getEOTSPrivKey(fpPk)

    // ❌ NO DATABASE CHECK - bypasses double-sign protection
    signedBytes, _ := eots.Sign(privKey, privRand, msg)
    return signedBytes, nil
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:C/C:C/A:C/I:C/D:C/Y:C (5.0)

Recommendation

- Add configuration option to disable `UnsafeSignEOTS` in production (default: disabled).
- Only enable when explicitly running in test/dev mode.
- Alternatively: Remove endpoint entirely from production builds.
- Consider build tags to exclude test endpoints from production binary.

Remediation Comment

SOLVED: The **Babylon Labs team** solved the issue in the specified commit.

Remediation Hash

https://github.com/babylonlabs-io/finality-provider/pull/736

7.3 TOCTOU RACE CONDITION IN SIGNEOTS AND SIGNBATCHEOTS

// LOW

Description

Both `SignEOTS` and `SignBatchEOTS` have Time-of-Check-Time-of-Use (TOCTOU) race conditions where concurrent calls can both read that a height has not been signed yet, then both sign the same height with different messages. This double-signing reveals the EOTS private key. Despite a mutex existing in the struct, neither function acquires the lock.

Impact:

- **EOTS key extraction**: Double-signing same height reveals private key mathematically.
- **Requires concurrent calls**: Exploiting race requires near-simultaneous calls to these functions.
- **Lower probability**: Race window is small (DB read to DB write), making exploitation timing-dependent.
- **Complete validator compromise**: If exploited, attacker can submit both signatures to slash validator.

Code Location

In `finality-provider/eotsmanager/localmanager.go`, the `LocalEOTSManager` struct:

Copy Code

```
36 | mu sync.Mutex  // ❌ Defined but NOT used in SignEOTS or SignBatchEOTS
```

In `finality-provider/eotsmanager/localmanager.go`, the `SignEOTS` function:

Copy Code

```
234 | func (lm *LocalEOTSManager) SignEOTS(eotsPk []byte, chainID []byte, msg []byte, height uint64) (*btcec.M
235 |     // ❌ NO LOCK ACQUIRED
236 |
237 |     // Line 235: Read existing record from DB (Time-of-Check)
238 |     record, found, err := lm.es.GetSignRecord(eotsPk, chainID, height)
239 |
240 |     if found {
241 |         if bytes.Equal(msg, record.Msg) {
242 |             return &s, nil  // Same message, return existing signature
243 |         }
244 |         return nil, eotstypes.ErrDoubleSign  // Different message, error
245 |     }
246 |
247 |     // Lines 267-291: Sign new height and save (Time-of-Use)
248 |     privRand, _, err := lm.getRandomnessPair(eotsPk, chainID, height)
249 |     privKey, err := lm.getEOTSPrivKey(eotsPk)
250 |     signedBytes, err := eots.Sign(privKey, privRand, msg)
251 |
252 |     // Line 287: Save to DB
253 |     if err := lm.es.SaveSignRecord(height, chainID, msg, eotsPk, b[:]); err != nil {
254 |         return nil, fmt.Errorf("failed to save signing record: %w", err)
255 |     }
256 | }
```

In `finality-provider/eotsmanager/localmanager.go`, the `SignBatchEOTS` function:

```
func (lm *LocalEOTSManager) SignBatchEOTS(req *SignBatchEOTSRequest) ([]SignDataResponse, error) {
    // ❌ NO LOCK ACQUIRED

    // Line 307: Read existing records from DB (Time-of-Check)
    existingRecords, err := lm.es.GetSignRecordsBatch(eotsPk, chainID, heights)

    // Lines 317-382: Loop through requests, sign new ones (Time-of-Use)
    for _, request := range req.SignRequest {
        if existingRecord, found := existingRecords[height]; found {
            continue  // Skip already signed
        }

        // Sign new height
        signedBytes, err := eots.Sign(privKey, privRand, msg)
        recordsToSave = append(recordsToSave, ...)
    }

    // Line 385: Save all new signatures to DB
    if err := lm.es.SaveSignRecordsBatch(recordsToSave); err != nil {
        return nil, err
    }
}
```

In `finality-provider/eotsmanager/localmanager.go`, other functions do use the mutex. Lines 440, 497, 530: other functions properly acquire `lm.mu.Lock()`.

BVSS

AO:S/AC:M/AX:L/R:N/S:U/C:C/A:C/I:C/D:C/Y:C (2.7)

Recommendation

- Acquire `lm.mu.Lock()` at the start of both `SignEOTS` and `SignBatchEOTS`, release with `defer lm.mu.Unlock()`.
- Follows existing locking pattern used in other functions in the same file.
- Simple one-line fix per function prevents race condition entirely.

Remediation Comment

SOLVED: The **Babylon Labs team** solved the issue in the specified commit.

Remediation Hash

https://github.com/babylonlabs-io/finality-provider/pull/726

7.4 ZOMBIE VERIFIED DELEGATIONS BREAK ECONOMIC SECURITY VIA VIGILANTE/BITCOIN RPC DOS

// INFORMATIONAL

Description

Vigilante is responsible for monitoring Bitcoin and reporting unbonding transactions to Babylon to remove voting power from unbonded delegations. Attackers can flood Babylon with zombie VERIFIED delegations (that pass validation but never exist on Bitcoin) for only Babylon gas fees, creating scalability and availability risk (increased activation latency and memory pressure) for Vigilante and Babylon infrastructure, but not a system-wide economic security failure under default settings.

1. **Startup vulnerability:** At initialization, Vigilante queries ALL delegations from Babylon (status ANY) including every accumulated zombie VERIFIED delegation. With deduplication checks, Vigilante only adds delegations not already in pendingTracker, but after every restart, the tracker is empty and must rebuild from ALL Babylon delegations. This means each restart re-processes all zombies, making restarts progressively slower and more resource-intensive as the VERIFIED set grows. The background sweep runs asynchronously, so restarts still complete, but upgrades and maintenance become increasingly costly for Vigilante operators. In addition, because `pendingTracker` retains full Bitcoin transactions for VERIFIED-without-proof delegations, a large zombie population can drive unbounded growth in this in-process structure and associated memory usage.

2. **Runtime vulnerability:** During normal operation, Vigilante uses event-driven updates for NEW delegations, mitigating steady-state impact. However, activation checking (runs on timer, queries Bitcoin RPC for each pending delegation) and unbonding detection (runs every Bitcoin block, queries Bitcoin indexer) share the same Bitcoin RPC client. Because the activation loop is single-threaded and sequential, zombies primarily increase the amount of work per scan and activation latency, and may modestly contend with other uses of the shared Bitcoin node; unbonding detection runs via the indexer and is not expected to fail system-wide under this pattern.

Critical consequence: The zombie VERIFIED pattern primarily increases activation latency and per-process resource usage in Vigilante. Under default settings, delegations without Bitcoin inclusion proofs do not carry voting power, and unbonding detection is driven by the indexer; the realistic risk is availability/UX degradation (delayed activation and unbonding processing on overloaded instances), not a systemic break of the economic security model.

Attack Scenario:

1. Attacker floods Babylon with zombie VERIFIED delegations (only Babylon gas cost, no Bitcoin transactions).
2. Vigilante queries ALL delegations, loads all zombies into pendingTracker.
3. Vigilante attempts Bitcoin RPC lookups for all zombies (all fail, waste time).
4. Bitcoin node and Vigilante experience increased load; activation processing falls behind.

5. Legitimate user unbonds ACTIVE delegation on Bitcoin.
6. On overloaded instances, Vigilante may be slower to detect/report unbonding.
7. During such delays, removal of voting power can be temporarily lagged on affected instances.
8. Overall, this scenario primarily reflects availability and UX risk rather than a guaranteed break of the economic security model.

Impact:

- **Availability / UX impact (Low severity)** – Increased activation latency and potential delays in unbonding processing on overloaded Vigilante instances; under default settings this does not by itself allow users to freely unbond while retaining voting power with zero slashing risk.
- **Reward misallocation window** – If unbonding processing is delayed on a given instance, there may be a temporary window where rewards are still computed against stake that is in the process of unbonding, until the system catches up.
- **Increased load across Vigilante instances** – All Vigilantes query the same Babylon state and must process zombie delegations, increasing CPU and memory usage and potentially degrading responsiveness, but not necessarily causing a full DoS.
- **Slow/expensive restarts** – Every Vigilante restart re-queries ALL delegations including zombies; with enough zombies, restart/upgrade becomes increasingly slow and resource-intensive, though the process still completes because the sweep runs in the background.
- **Cascading latency** – Large zombie sets can slow delegation activation (VERIFIED→ACTIVE) and, on overloaded instances, may delay unbonding detection (ACTIVE→UNBONDED), impacting timeliness rather than correctness.
- **Babylon query DoS** - Accumulation of invalid VERIFIED delegations causes query timeouts → explorers and third-party apps fail.
- **Genesis export failure** - Too many VERIFIED delegations prevent chain migration and upgrades.
- **Resource exhaustion** – Unbounded growth of pendingTracker as it stores full Bitcoin transactions for many VERIFIED-without-proof delegations can lead to significant per-process memory pressure and potential OOM on Vigilante instances; Bitcoin node RPC usage increases due to repeated failed lookups, but the single-threaded scan keeps QPS relatively low.
- **Permanent storage pollution** - Invalid delegations never deleted, causing unbounded Babylon state growth.
- **Low attack cost** - Only Babylon transaction fees, no Bitcoin fees required, economically viable due to flat gas pricing.
- **Fast execution** - Attacker can create zombies faster than Vigilante can process them (Babylon 6-second blocks vs Bitcoin RPC latency).

Code Location

In `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go`, Vigilante queries ALL delegations at startup:

Copy Code

```
184 go sew.fetchDelegations()  // Line 184 - called ONCE at startup
185
186 func (sew *StakingEventWatcher) checkBabylonDelegations() error {  // Line 273
```

```
    status := btcstakingtypes.BTCDelegationStatus_ANY  // Line 274 - queries ALL
    delegations, nextCursor, err := sew.babylonNodeAdapter.DelegationsByStatus(status, cursor, limit)  /

    case btcstakingtypes.BTCDelegationStatus_VERIFIED.String():  // Line 290
        sew.addToPendingFunc(delegation)  // ← Adds ALL VERIFIED including zombies
}

func (sew *StakingEventWatcher) addToPendingFunc(delegation Delegation) {  // Line 1096
    _, exists := sew.pendingTracker.GetDelegation(stkTxHash)  // Line 1106
    if !exists && !delegation.HasProof {  // Line 1107 - deduplication check
        sew.pendingTracker.AddDelegation(...)  // Only adds if not already tracked
    }
}
// After startup: event-driven updates only track NEW delegations (lines 998-1036)
// But EVERY restart re-queries ALL delegations including accumulated zombies
```

Activation checking runs on timer:

Copy Code

```
ticker := time.NewTicker(sew.cfg.CheckDelegationsInterval)  // Line 686
case <-ticker.C:  // Line 691
    sew.checkBtcForStakingTx()  // Line 693

func (sew *StakingEventWatcher) checkBtcForStakingTx() {  // Line 704
    for del := range sew.pendingTracker.DelegationsIter(1000) {
        details, status, err := sew.btcClient.TxDetails(&txHash, del.StakingTx.TxOut[del.StakingOutputId
        // ← NO RATE LIMITING on Bitcoin RPC calls (line 711)
        // activationLimiter only applies to submitting inclusion proofs (line 741), NOT lookups
    }
}
```

Unbonding detection runs on every Bitcoin block:

Copy Code

```
case block := <-sew.btcNotifier.Blocks():  // Line 244
    if err := sew.checkSpend(); err != nil {  // Line 257 - called on EVERY block
        sew.logger.Errorf("error checking spend: %v", err)
    }

func (sew *StakingEventWatcher) checkSpend() error {  // Line 535
    for del := range sew.unbondingTracker.DelegationsIter(1000) {  // Line 536
        response, err := sew.indexer.GetOutspend(ctx, del.StakingTx.TxHash().String(), del.StakingOutput
        // ← Independent tracker, but shares same Bitcoin RPC client
    }
}
```

However, this loop is single-threaded and sequential, so the effective RPC QPS is low; the primary concern is cumulative latency and per-process resource usage when pendingTracker is large, not a high-volume RPC DoS.

Vigilante reports unbonding to remove voting power:

Copy Code

```
err = sew.babylonNodeAdapter.ReportUnbonding(ctx, stakingTxHash, stakeSpendingTx, proof, fundingTxs)
// Sends MsgBTCUndelegate to Babylon
```

In babylon/x/btcstaking/keeper/btc_delegations.go, Babylon removes voting power ONLY when Vigilante reports:

```
func (k Keeper) btcUndelegate(ctx sdk.Context, btcDel *types.BTCDelegation, u *types.DelegatorUnbondingI
    if !btcDel.HasInclusionProof() {
        return  // VERIFIED delegations have no power anyway
    }
    // ACTIVE delegations: remove voting power
    unbondedEvent := types.NewEventPowerDistUpdateWithBTCDel(event)
    k.addPowerDistUpdateEvent(ctx, btcTip.Height, unbondedEvent)
}
```

Babylon accepts delegations without Bitcoin transactions:

```
if !parsedMsg.IsIncludedOnBTC() {
    ctx.GasMeter().ConsumeGas(params.DelegationCreationBaseGasFee, "delegation creation fee")
}
// Accepts delegation even if never broadcast to Bitcoin
```

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

1. **Babylon**: Implement garbage collection for VERIFIED delegations without inclusion proof after timeout period (e.g., 1 week).
2. **Babylon**: Add rate limiting on delegation creation per account/block.
3. **Vigilante**: Query only recent VERIFIED delegations (e.g., created within last N blocks) instead of ALL.
4. **Vigilante**: Prioritize unbonding detection over delegation activation to preserve economic security.
5. **Vigilante**: Add monitoring/alerting for abnormal VERIFIED delegation growth.

Remediation Comment

ACKNOWLEDGED: The **Babylon Labs team** acknowledged this finding, mentioning that:

> Babylon gas costs should be enough to avoid attackers to create multiple delegations as to include an transaction without the BTC inclusion proof we charge extra gas.

7.5 VALIDATION GAP: FUNDING OUTPUT VALUE

// INFORMATIONAL

Description

Babylon relies on indirect validation (amount increase and fee checks) to reject expansions with non-positive funding values, while covenant has an explicit check. If Babylon's indirect checks have a bug, users' funds could get stuck in `PENDING` state.

Impact:

- If Babylon's indirect checks have a logic bug, invalid expansion could be accepted
- Covenant would then reject it (no signatures provided)
- **Result**: Delegation stuck in PENDING state forever, user funds locked
- If Babylon validation works correctly: No impact (covenant check is redundant defense-in-depth)

Code Location

In `babylon/x/btcstaking/keeper/msg_server.go` , funding value is read without explicit validation:

```
744 | fundingInputValue := parsedMsg.StkExp.OtherFundingOutput.Value
745 | totalInputValue := oldStakingAmt + fundingInputValue
```

Amount increase check (indirect protection):

```
739 | if newStakingAmt < oldStakingAmt {
740 |     return fmt.Errorf("staking expansion output amount %d is less than previous delegation amount %d", .
741 | }
```

Fee validation (indirect protection):

```
739 | impliedFee := totalInputValue - totalOutputValue
740 | if impliedFee <= 0 {
741 |     return fmt.Errorf("invalid transaction fee: inputs %d <= outputs %d", ...)
742 | }
```

In `covenant-emulator/covenant/covenant.go` , explicit validation:

```
378 | if otherOutput.Value <= 0 {
379 |     return nil, fmt.Errorf("funding output has invalid value %d", otherOutput.Value)
380 | }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

- Add explicit funding value check in Babylon: `if fundingInputValue <= 0 { return error }`.
- Makes critical constraint explicit rather than relying on indirect validation.
- Prevents potential stuck PENDING delegations if indirect checks have logic bugs.
- Aligns validation approach with covenant for consistency.

Remediation Comment

SOLVED: The **Babylon Labs team** solved the issue in the specified commit.

Remediation Hash

https://github.com/babylonlabs-io/babylon/pull/1855

7.6 STALE PARAMETER AFTER GOVERNANCE UPDATE

// INFORMATIONAL

Description

Vigilante caches three governance parameters at startup (`BtcConfirmationDepth` , `CheckpointFinalizationTimeout` , `CheckpointTag`) and never refreshes them. When these parameters are updated via governance proposals, vigilante continues using stale cached values, causing rejected proofs and wasted transaction fees.

Impact:

- Rejected proof submissions leading to wasted transaction fees when parameter values increase
- Increased delegation activation latency as vigilante retries with stale confirmation depths
- Operational inefficiency until vigilante is manually restarted
- No security risk: Babylon validates with current parameters, ensuring correctness (self-correcting behavior)

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

- Query governance parameters periodically (e.g., hourly) instead of caching at startup only
- Subscribe to governance parameter change events and refresh cached values
- Workaround: Restart vigilante manually after governance parameter updates

Remediation Comment

ACKNOWLEDGED: The **Babylon Labs team** acknowledged this finding, mentioning that:

> That is expected, we will instruct vigilante operators to restart if we ever update those gov parameters `BtcConfirmationDepth` , `CheckpointFinalizationTimeout` , `CheckpointTag` .

Remediation Hash

https://github.com/babylonlabs-io/vigilante

7.7 UNUSED CONTEXT PARAMETER PREVENTS CANCELLATION

// INFORMATIONAL

Description

`IsFPSlashed` receives caller's context but ignores it. Babylon client internally creates its own context with timeout (protecting against indefinite hangs), but caller's cancellation signals are not respected.

Code locations:

- `vigilante/btcstaking-tracker/atomicslasher/babylon_adapter.go:115` - Context parameter ignored.
- `babylon/client/query/client.go:73-79` - New context created from `Background()` instead of using parent.

Impacts:

- Caller cannot cancel ongoing RPC calls (e.g., during shutdown).
- May delay vigilante shutdown by up to configured client timeout duration.
- Timeout protection EXISTS (client creates context with timeout), preventing indefinite hangs.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Update Babylon client to accept parent context and derive timeout context from it.
- Change `context.WithTimeout(context.Background(), timeout)` to `context.WithTimeout(parentCtx, timeout)`.

Remediation Comment

FUTURE RELEASE: The **Babylon Labs team** acknowledged the issue and plans to fix it in the future.

7.8 INCONSISTENT ERROR LOGGING CONTEXT

// INFORMATIONAL

Description

Vigilante

Some error logs in vigilante are missing the delegation identifier (`staking_tx_hash`), making it difficult for operators to correlate operational failures to specific delegations during troubleshooting.

Impact:

- **Debugging difficulty**: Operators cannot identify which delegation caused the error
- **Operational overhead**: Must correlate logs with other events to troubleshoot issues
- **Inconsistent logging**: Some errors include context, others don't

Code Location

In `vigilante/btcstaking-tracker/stakingeventwatcher/stakingeventwatcher.go` , log WITH delegation identifier:

```
713 | sew.logger.Errorf("error getting tx: %v, err: %v", txHash, err)
```

Log WITHOUT delegation identifier:

```
663 | sew.logger.Errorf("error adding delegation to unbondingTracker: %v", err)
664 | // ❌ Missing: activeDel.stakingTxHash
```

Finality Provider

Error log in finality provider is missing the finality provider identifier (BTC public key), making it difficult for operators to determine which FP instance encountered the error when running multiple FP instances.

Impact:

- **Multi-instance debugging**: Operators running multiple FP instances cannot identify which instance failed.
- **Critical error tracking**: Channel full errors are critical but lack instance context.
- **Operational overhead**: Must correlate with other logs to identify the FP.

Code Location

In `finality-provider/finality-provider/service/fp_instance.go` , Log WITH FP identifier:

```
fp.logger.Warn("the finality-provider is jailed",
    zap.String("pk", fp.GetBtcPkHex()),
)
```

Log WITHOUT FP identifier:

```
fp.logger.Error("failed to report critical error (channel full)", zap.Error(err))
// ❌ Missing: zap.String("fp_btc_pk", fp.GetBtcPkHex())
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add `staking_tx_hash` to error log:

```
sew.logger.Errorf("error adding delegation to unbondingTracker: staking_tx_hash=%s, err: %v",
activeDel.stakingTxHash, err)
```

Add `fp_btc_pk` to error log:

```
fp.logger.Error("failed to report critical error (channel full)",
zap.String("fp_btc_pk", fp.GetBtcPkHex()),
zap.Error(err))
```

Remediation Comment

SOLVED: The **Babylon Labs team** solved this issue.

Remediation Hash

https://github.com/babylonlabs-io/vigilante/pull/482, https://github.com/babylonlabs-io/finality-provider/pull/737

7.9 INCONSISTENT VALIDATION ERROR CONTEXT

// INFORMATIONAL

Description

Some validation error logs are missing the delegation identifier (`staking_tx_hex`), making it difficult for operators to correlate validation failures to specific delegations during troubleshooting.

Impact:

- Operators cannot correlate validation failures to specific delegations in logs.
- During incidents, operators waste time trying to identify which user's delegation failed.
- Inconsistent logging format makes automated log parsing and alerting more difficult.

Code Location

1. **Logs with delegation idenfifier**:

In `covenant-emulator/covenant/covenant.go` , empty undelegation check:

```
103 | ce.logger.Debug("empty undelegation",
104 |     zap.String("staking_tx_hex", btcDel.StakingTxHex))
```

Covenant signatures fulfilled check:

```
119 | ce.logger.Debug("covenant signatures already fulfilled",
120 |     zap.String("staking_tx_hex", btcDel.StakingTxHex))
```

Invalid delegation check:

```
168 | ce.logger.Debug("invalid delegation",
169 |     zap.String("staking_tx_hex", btcDel.StakingTxHex))
```

2. **Logs without delegation identifier:**

Invalid unbonding time:

```
130 | ce.logger.Error("invalid unbonding time",
131 |     zap.Uint32("expected_unbonding_time", unbondingTimeBlocks),
132 |     zap.Uint16("got_unbonding_time", unbondingTime),
133 |     // ❌ Missing: zap.String("staking_tx_hex", btcDel.StakingTxHex)
134 | )
```

Invalid staking time:

```
ce.logger.Error("invalid staking time",
    zap.Uint16("min_staking_time", params.MinStakingTime),
    zap.Uint16("max_staking_time", params.MaxStakingTime),
    zap.Uint16("got_staking_time", stakingTime),
    // ❌ Missing: zap.String("staking_tx_hex", btcDel.StakingTxHex)
)
```

Invalid staking value:

```
ce.logger.Error("invalid staking value",
    zap.Int64("min_staking_value", int64(params.MinStakingValue)),
    zap.Int64("max_staking_value", int64(params.MaxStakingValue)),
    zap.Int64("got_staking_value", int64(btcDel.TotalSat)),
    // ❌ Missing: zap.String("staking_tx_hex", btcDel.StakingTxHex)
)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Add `zap.String("staking_tx_hex", btcDel.StakingTxHex)` to all validation error logs.
- Standardize logging format across all validation checks for consistency.
- Consider adding additional context fields (delegation height, params version) for better debugging.

Remediation Comment

SOLVED: The **Babylon Labs team** solved this issue in the specified commit.

Remediation Hash

https://github.com/babylonlabs-io/covenant-emulator/pull/165

7.10 INCORRECT METRIC TYPES (GAUGE VS COUNTER)

// INFORMATIONAL

Description

Two finality provider metrics use Prometheus Gauge type instead of Counter type for cumulative counts, causing incorrect semantics for monitoring tools.

Impact:

- **Wrong semantics**: Gauge is for fluctuating values, Counter is for monotonically increasing values.
- **Monitoring confusion**: Counters support `rate()` / `increase()` functions; Gauges use `delta()` / `deriv()`.
- **Dashboard issues**: May confuse monitoring interpretations and alerting rules.
- **Functionally works**: Both types support `.Inc()` and `.Add()`.

Code Location

In `finality-provider/metrics/fp_collectors.go`, `fpTotalVotedBlocks` — used with `.Inc()` and `.Add()` at lines 231, 236 (counter behavior).

Copy Code

```
106 fpTotalVotedBlocks: prometheus.NewGaugeVec(
107     prometheus.GaugeOpts{  // ❌ Should be CounterOpts
108         Name: "fp_total_voted_blocks",
109         Help: "The total number of blocks voted by a finality provider.",
110     },
111     []string{"fp_btc_pk_hex"},
112 ),
```

`fpTotalCommittedRandomness` — used with `.Add()` at line 241 (counter behavior).

Copy Code

```
113 fpTotalCommittedRandomness: prometheus.NewGaugeVec(
114     prometheus.GaugeOpts{  // ❌ Should be CounterOpts
115         Name: "fp_total_committed_randomness",
116         Help: "The total number of randomness commitments by a finality provider.",
117     },
118     []string{"fp_btc_pk_hex"},
119 ),
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Change from `GaugeVec` to `CounterVec`:

```
fpTotalVotedBlocks: prometheus.NewCounterVec(
prometheus.CounterOpts{  // Changed from GaugeOpts
    Name: "fp_total_voted_blocks",
    Help: "The total number of blocks voted by a finality provider.",
},
[]string{"fp_btc_pk_hex"},
),

fpTotalCommittedRandomness: prometheus.NewCounterVec(
prometheus.CounterOpts{  // Changed from GaugeOpts
    Name: "fp_total_committed_randomness",
    Help: "The total number of randomness commitments by a finality provider.",
},
[]string{"fp_btc_pk_hex"},
),
```

Remediation Comment

SOLVED: The **Babylon Labs team** solved this issue in the specified commit, replacing the `GaugeVec` occurences by `CounterVec` .

Remediation Hash

https://github.com/babylonlabs-io/finality-provider/pull/735

7.11 UNUSED CONFIGURATION VARIABLE: NUMPUBRANDMAX

// INFORMATIONAL

Description

The `NumPubRandMax` configuration field is defined with a default value of 500,000 but is never validated or used anywhere in the codebase, despite having a description suggesting it should limit randomness generation.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

- Option 1: Remove unused `NumPubRandMax` field from configuration.
- Option 2: Implement intended validation logic to enforce maximum randomness generation limit.
- If implementing: Add validation in EOTS Manager `CreateRandomnessPairList` to reject requests exceeding `NumPubRandMax`.

Remediation Comment

SOLVED: The **Babylon Labs team** solved this issue.

Remediation Hash

https://github.com/babylonlabs-io/finality-provider/pull/734

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.