Lombard Approver

Security Review

Cantina Managed review by:
Bernd, Security Researcher

Dontonka, Associate Security Researcher
Haxatron, Associate Security Researcher

January 22, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Lombard is on a mission to expand the digital economy by transforming Bitcoin's utility from a store of value into a productive financial tool with LBTC.

From Nov 26th to Dec 8th the Cantina team conducted a review of Lombard-approver on commit hash d9b23109. The team identified a total of **35** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	1	1	0
Medium Risk	10	10	0
Low Risk	9	9	0
Gas Optimizations	5	5	0
Informational	9	9	0
Total	**35**	**35**	**0**

3 Findings

3.1 Critical Risk

3.1.1 Unstake requests can be griefed by forged and yet unconfirmed notary sessions

Severity: Critical Risk

Context: unstake.go#L184-L187

Description: Notary sessions can be permissionlessly submitted to the Lombard ledger through the `MsgSubmitPayload` Cosmos SDK message (msg_server_submit_payload.go#L13-L41). Although a quorum of authorized notaries is needed for confirmation, any notary session, unconfirmed or confirmed, is retrieved by the approver and processed thereafter.

When processing an unstake receipt, the associated unstake request will be marked as paid (unstake.go#L185) in `processReceipt()`, so long as the receipt is not expired yet. This measure is intended as a safety guard to prevent multiple payouts for the same unstake request.

However, by prematurely marking an unstake request as paid, the approver will reject it, as it cannot match the UTXO output with an unpaid unstake request. As a result, it errors (lombard_unstake.go#L139) with `"no valid unpaid unstake notarization found on Ledger"` in `validateUnstakingOutput()`.

Consequently, forged unstake receipts can be used to grief users by preventing them from receiving their unstaked BTC.

Recommendation: When processing unstake receipts in `processReceipt()`, consider only marking the associated unstake request as paid if the receipt has the status `notarytypes.Completed`, i.e., it has been confirmed by the required quorum of notaries.

Lombard: Fixed in PR 72.

Cantina Managed: Fix looks good

3.2 High Risk

3.2.1 Lombard unstake functionality currently not working as expected

Severity: High Risk

Context: lombard_unstake.go#L67-L112

Description: `validateUnstakingOutputs` is too strict and currently DOS the whole Lombard unstake functionality.

A recent mitigation (commit-SHA `1fac8baea`) seems to have broken Lombard unstake as a whole, as not allowing any room for the **miner fee** during the validation process of unstake MFA request, which should result in almost all the Lombard unstake request to be rejected (lombard_unstake.go#L28), assuming here that the proposer would be proposing "*valid*" request which include a miner fee. If this assumption would be wrong, that would also be an important negative impact (but no fund loss, once the issue is fixed those transactions can be re-submitted), as transaction would be broadcasted to BTC, but since those doesn't include miner fee would likely remain stuck in the mempool indefinitely, preventing the user to get his BTC back.

A BTC UTXO is structured as follow:

- Input(s) to spent (which point to an output from a previous UTXO).
- Output(s) (which need to cover fully this Input(s) - fee for miner).
  - Output (Part of the input).
  - Change (The remaining part minus fees).

So the sums of outputs values must be less then total of inputs, as what remains is the **miner fee**, the cost to include this transaction in the block. In the current function, there is no room for the fees (if `totalOut != totalIn`; lombard_unstake.go#L108).

Here is a typical example.

```
Previous Transaction:
Output 0: 1 BTC to Address A
Output 1: 2 BTC to Address B

New Transaction:
If you want to spend Output 0 (1 BTC), you must use all of it:
- You can't just use 0.5 BTC from it
- If you only want to send 0.5 BTC to someone, you need to:
  1. Use the entire 1 BTC input
  2. Create two outputs:
     - 0.5 BTC to recipient
     - ~0.499 BTC back to yourself as change (minus transaction fee)
```

Recommendation: Allow some room for the miner fee.

Lombard: Fixed in PR 78.

Cantina Managed: Fix looks good.

3.3 Medium Risk

3.3.1 Error handling in `BulkCreateUnstakes` could DOS the notary session processing as a whole

Severity: Medium Risk

Context: unstake.go#L86-L94

Description: Error handling in `BulkCreateUnstakes` could DoS the notary session processing as a whole. Unstake requests and their receipts are processed in the following way by the processor service. Whenever unstakes are returned from the ledger service (`len(unstakes) > 0`; unstake.go#L83), database service `BulkCreateUnstakes` is called to create/update the DB accordingly.

`processUnstakeNotarizations` → `processBatch` → `BulkCreateUnstakes`

In the low probability that any of the following errors are detected (DB corruption, etc.), the function will hard stop and return an error which will bubble up to the service (service.go#L67-L68), preventing any unstake to be created or updated in the DB.

- unstake.go#L87-L89: `ScriptPubKey` (the recipient) mismatch.
- unstake.go#L92-L94: Amount mismatch.

Since those transactions will always keep coming back (as they are not deleted from the DB and/or ledger), this will effectively result in the denial-of-service of the notary session processing as a whole.

Recommendation: Whenever such errors are detected, log an error trace, skip it, and continue processing. That will allow the processing to continue without much harm while allowing you to take manual action to resolve this problematic situation.

```
  // An unstake with this ID exists, check if script pubkey matches
  if !bytes.Equal(existingUnstake.ScriptPubKey, unstake.ScriptPubKey) {
+      d.logger.Errorf("conflict detected: unstake with ID %d exists with different script pub key", unstake.I
D)
+      continue
-      return errors.Errorf("conflict detected: unstake with ID %d exists with different script pub key",
↪  unstake.ID)
  }

  // An unstake with this ID exists, check if amount match
  if existingUnstake.Amount != unstake.Amount {
+      d.logger.Errorf("conflict detected: unstake with ID %d exists with different amount", unstake.ID)
+      continue
-      return errors.Errorf("conflict detected: unstake with ID %d exists with different amount", unstake.ID)
  }
```

Lombard: Fixed in PR 51, specifically commit ebb92f9a.

Cantina Managed: Fix looks good.

3.3.2 No-mix staking wallets are not fully handled in the codebase

Severity: Medium Risk

Context: common.go#L117-L125, common.go#L191-L199, common.go#L226-L235

Description: In the Lombard codebase, there are two types of staking wallets - the regular staking wallet and the no-mix staking wallet.

The problem is that the no-mix staking wallets are not handled in numerous parts of the codebase. In all of these instances, it only checks that the public key is the regular staking wallet. If it is the no-mix staking wallet, it will return an error.

1. `validateStakingInputs`: This function is called in Babylon deposit and Lombard unstake, which means that it would actually be impossible to deposit to Babylon or withdraw funds from the no-mix staking wallet.
2. `validateStakingOutputs`: This function is called in Lombard transfer, meaning it would be impossible to transfer funds from the deposit wallet to the no-mix staking wallet from regular UTXOs. Notably, it is still possible via the change UTXO.
3. `validateStakerKey`: This function is called in Babylon early unbond and timelock withdraw, which means it would be impossible to perform an early unbonding request or withdraw from Babylon to the no-mix staking wallet.

In the current implementation, as elaborated earlier, funds can still enter the no-mix staking wallet via the change UTXO, so it is possible for funds to become stuck there.

Recommendation: The no-mix staking wallets should also be checked in these parts of the codebase

Lombard: Fixed in PR 67, specifically commit 37747e76. To make the solution less code-redundant, we merged the functions to fetch the main staking key and fetch no-mix staking key into one function that tries to fetch it from both. This will work because at most 1 of the sessions has access to this key. I also added the checks for whether the key is enabled into this function.

Cantina Managed: Fix looks good.

3.3.3 Invalid public keys can cause the approver service to panic and crash

Severity: Medium Risk

Context: common.go#L105-L118, common.go#L179-L192, common.go#L305-L316, custodian_-manager.go#L188-L200

Description: In multiple instances of the codebase, it was observed that this same code was used:

```go
_, addrs, numAddrs, err := txscript.ExtractPkScriptAddrs(pkScript, c.blockchainService.GetNetwork())
if err != nil {
    return errors.Wrap(err, "error extracting addresses from pubkey script for PSBT input")
}
,
// Ignore non-standard or null scripts (0 addresses)
// Ignore multisig addresses because staking and deposit wallets are not multisig
if numAddrs != 1 {
    return errors.Errorf("invalid number (%d) of addresses extracted from pubkey script for PSBT input",
↪   numAddrs)
}

inputAddr := addrs[0].EncodeAddress()
```

This code is problematic as it can cause the approver service to panic and crash if an invalid public key is submitted. This is because during `ExtractPkScriptAddrs`, an invalid public key will result in an `err` in `btcutil.NewAddressPubKey` function, which can cause the `addrs` array to be empty while the `numAddrs` to be 1.

- txscript/standard.go#L966-L991:

```go
// ExtractPkScriptAddrs returns the type of script, addresses and required
// signatures associated with the passed PkScript.  Note that it only works for
// 'standard' transaction script types.  Any data such as public keys which are
// invalid are omitted from the results.
func ExtractPkScriptAddrs(pkScript []byte,
    chainParams *chaincfg.Params) (ScriptClass, []btcutil.Address, int, error) {
    // ...
    // Check for pay-to-pubkey script.
    if data := extractPubKey(pkScript); data != nil {
        var addrs []btcutil.Address
        addr, err := btcutil.NewAddressPubKey(data, chainParams)
        if err == nil {
            addrs = append(addrs, addr)
        }
        return PubKeyTy, addrs, 1, nil
    }
    // ...
```

As a result, the code will reach `inputAddr := addrs[0].EncodeAddress()`, which will result in access out-of-bounds of the `addrs` array, which will lead to the approver service panic and crash.

Recommendation: Also check that `len(addrs) != 1` and return error if it is.

Lombard: Fixed in PR 54, specifically commit aa6312ec

Cantina Managed: Fix looks good.

3.3.4 Not all staking key enabled status are checked

Severity: Medium Risk

Context: common.go#L331-L337, custodian_manager.go#L201-L220

Description: The staking key enabled status is not checked in multiple places. The approver service checks that the staking key is enabled before accepting it, as in the event of a key compromise. This allows the Lombard team to disable such a compromised key via Cubist, which will automatically prevent the approver service from approving such keys.

The problem arises due to the fact that the approver does not sufficiently check that the key is enabled in numerous places.

Recommendation: Consider checking the enabled status in all the highlighted instances.

Lombard: Fixed in PR 62 and PR 75 (the latter ensures that if fetching the key from the first session returns an invalid result; no error, but key is either nil or disabled; we will return early with that error).

Cantina Managed: Shouldn't client.go#L140-L142 return `noMixStakingRoleID` instead of `stakingRoleID`?

Lombard: Definitely. Fixed in commit e781946b.

Cantina Managed: Fix looks good.

3.3.5 Approver cannot handle "no mix" Cubist MFA signing requests that require the `noMixClientSession` session

Severity: Medium Risk

Context: internal/cubist/service.go

Description: Lombard might segregate institutional funds from retail funds by using separate staking wallets, "no mix" and regular staking wallets. In addition, separate Cubist keys/roles are used to completely separate the two types of funds.

For this to work, the approver uses two Cubist sessions, `noMixClientSession` and `clientSession`. It is expected that MFA signing requests that are associated with "no mix" funds will only be retrieved using the `noMixClientSession` while all other requests will be retrieved using the `clientSession`. In the same way, MFA requests must be approved and rejected using the corresponding session.

However, the approver currently only uses the `clientSession` for all MFA requests. The `noMixClientSession` is only used in `GetNoMixStakingKey()` (service.go#L136) to retrieve the staking key for the "no mix" funds. Consequently, the approver is not able to properly handle and process MFA requests that are separate from the general retail funds, and that would require the `noMixClientSession` Cubist session.

Recommendation: Consider selecting the appropriate Cubist session depending on the type of funds associated with the MFA request.

Lombard: Fixed in PR 67; specifically commit 6b973414. The solutions for List/Approve/Reject MFA request are different from Get Staking Key, namely:

- `ListMfaRequests`: We want to an aggregate of all MFA requests this user has access to. For use in approvals/rejections later, I track each MFA request with its corresponding session.
- `ApproveMfaRequest`, `RejectMfaRequest`: We want to approve/reject the MFA request for its corresponding session.
- `GetMfaRequest`: I noticed that we were calling this function almost immediately after `ListMfaRequests`; in my opinion this is not necessary as 1) it is unlikely that the state of the request has changed and 2) `GetMfaRequest`'s response doesn't provide any additional data.

Cantina Managed: Fix looks good.

3.3.6 Missing validation that ensures unspent BTC is fully sent back as change in Lombard transfer signing strategy

Severity: Medium Risk

Context: common.go#L166-L203

Description: The Lombard transfer signing strategy is used to transfer BTC from the Cubist deposit wallets to the Cubist staking hot wallets. `validateLombardTransferRequest()` ensures that the transfer request is valid before approving it.

However, there is no validation that ensures that the BTC miner fee is reasonable and that the remaining unspent BTC is sent back as change to one of Lombard's staking addresses. This could lead to a situation where the UTXO outputs only spend a fraction of the input BTC, with the remaining BTC going to the miner.

Recommendation: Consider summing the UTXO input and output values to determine the unspent BTC, which should be returned as change after deducting a reasonable miner fee.

Lombard: Fixed in PR 78.

Cantina Managed: Shouldn't `validateDepositInputs()` also validate `in.WitnessUtxo.Value == int64(txOut.Value*1e8)`?

Lombard: Added in commit bc8274f6.

Cantina Managed: Fix looks good.

3.3.7 Failures in `processReceipt` may DoS the entire processing

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: During `processBatch`, if there are any errors encountered in `processReceipt`, then `len(receipts)-receiptSucessCount` will be greater than 0. This will cause the error to bubble up after the end of the batch to `processUnstakeNotarizations`

- internal/processor/unstake.go#L92-L106:

```go
var receiptSucessCount int
for _, receipt := range receipts {
    if err := s.processReceipt(ctx, receipt); err != nil {
        s.logger.WithError(err).Errorf("failed to process unstake receipt %d", receipt.ID)
        continue
    }
    receiptSucessCount++
}

s.logger.Infof("processed %d unstake receipts", receiptSucessCount)

if len(receipts)-receiptSucessCount > 0 {
    return errors.Errorf("failed to process %d unstake receipts", len(receipts)-receiptSucessCount)
}
```

If there are more than `s.config.PageSize` receipts that will return an error, then it is possible to entirely DOS the processing. This is because it will be stuck processing this batch of receipts. Since all of them return an error, it will be stuck here.

- internal/processor/unstake.go#L15-L35:

```go
func (s *service) processUnstakeNotarizations(ctx context.Context) error {
    limit := s.config.PageSize
    offset, err := s.databaseService.FindOldestPendingID(ctx)
    if err != nil {
        return errors.Wrap(err, "error finding oldest pending ID of unstakes and receipts")
    }

    for {
        notarySessions, err := s.ledgerService.ListNotarySessions(ctx, offset, limit)
        if err != nil {
            return errors.Wrap(err, "error listing notary sessions from ledger")
        }

        if len(notarySessions.NotarySessions) == 0 {
            s.logger.Info("no more notarizations to process")
            return nil
        }

        if err := s.processNotarySessionBatch(ctx, notarySessions.NotarySessions); err != nil {
            return err
        }
```

Recommendation: Do not return `err` if `len(receipts)-receiptSucessCount > 0`, instead consider logging a warning.

Lombard: Fixed in PR 58; specifically commit 05cee6b6.

Cantina Managed: Fix looks good.

3.3.8 All notary sessions starting from offset 0 are repeatedly processed when there are no pending unstakes or unstake receipts

Severity: Medium Risk

Context: unstake.go#L48

Description: `FindOldestPendingID()` retrieves the oldest pending unstake or unstake receipt ID from the database.

```go
func (d *gormDriver) FindOldestPendingID(ctx context.Context) (uint64, error) {
    pending := strings.ToLower(notarytypes.Pending.String())

    var result struct {
        ID uint64
    }

    query := `
        SELECT MIN(id) as id FROM (
            SELECT id FROM unstakes WHERE state = ?
            UNION
            SELECT id FROM unstake_receipts WHERE state = ?
        )
    `

    err := d.db.WithContext(ctx).Raw(query, pending, pending).Scan(&result).Error
    if err != nil {
        return 0, err
    }

    return result.ID, nil
}
```

This ID is subsequently used as the pagination offset when querying the notary sessions (unstake.go#L23) from the `ledger` in `processUnstakeNotarizations()`.

However, if there are no pending unstakes or receipts, the function returns `result.ID`, which is `0`. Consequently, the approver will repeatedly process all notary sessions from the ledger, starting from offset 0.

This situation is temporarily resolved as soon as a pending unstake or unstake receipt is received and processed. However, when it is completed again, the issue will reappear. Over time, this can lead to performance degradation and increased processing time as the amount of notary sessions grows, preventing timely unstakes.

Recommendation: Consider explicitly keeping track of the last processed pending ID. This allows the approver to correctly resume processing notary sessions from the ledger, even when there are no pending unstakes or receipts in the database.

Lombard: Fixed in PR 81 which also contains a fix for "Potential data inconsistency caused by errors during unstake receipt processing" since both findings are somewhat related.

Cantina Managed: Fix looks good.

3.3.9 Incorrect conversion of covenant public key from Babylon API

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When processing the Babylon global params, the approver calls this function on the covenant public keys:

- internal/babylon/utils.go#L68-L89:

```go
func parseCovenantKeysFromSlice(covenantMembersPks []string) ([]*btcec.PublicKey, error) {
    covenantPubKeys := make([]*btcec.PublicKey, len(covenantMembersPks))

    for i, fpPk := range covenantMembersPks {
        fpPkBytes, err := hex.DecodeString(fpPk)
        if err != nil {
            return nil, err
        }

        if len(fpPkBytes) == 33 && (fpPkBytes[0] == 0x02 || fpPkBytes[0] == 0x03) {
            fpPkBytes = fpPkBytes[1:]
        }
        fpSchnorrKey, err := schnorr.ParsePubKey(fpPkBytes)
        if err != nil {
            return nil, err
        }

        covenantPubKeys[i] = fpSchnorrKey
    }

    return covenantPubKeys, nil
}
```

As can be seen, this function does two things - First, it checks the length of the key to ensure it is a valid compressed Secp256k1 public key and removes the first byte, the format byte. It then passes this to the function `schnorr.ParsePubKey`. Internally, this function appends the `PubKeyFormatCompressedEven` format byte (0x02) and then passes it to `btcec.ParsePubKey`:

- btcec/schnorr/pubkey.go#L23-L41:

```go
func ParsePubKey(pubKeyStr []byte) (*btcec.PublicKey, error) {
    if pubKeyStr == nil {
        err := fmt.Errorf("nil pubkey byte string")
        return nil, err
    }
    if len(pubKeyStr) != PubKeyBytesLen {
        err := fmt.Errorf("bad pubkey byte string size (want %v, have %v)",
            PubKeyBytesLen, len(pubKeyStr))
        return nil, err
    }

    // We'll manually prepend the compressed byte so we can re-use the
    // existing pubkey parsing routine of the main btcec package.
    var keyCompressed [btcec.PubKeyBytesLenCompressed]byte
    keyCompressed[0] = secp.PubKeyFormatCompressedEven
    copy(keyCompressed[1:], pubKeyStr)

    return btcec.ParsePubKey(keyCompressed[:])
}
```

For a compressed Secp256k1 public key, the format byte can only be one of two values, 0x02 or 0x03, and this byte is important in determining which y-coordinate on the elliptic curve to use when parsing the rest of the 32 bytes of the public key (corresponding to the x-coordinate).

The problem here is that covenant public keys can use both formats, but the approver will incorrectly decode the public key as if it is using a 0x02 format byte. This results in the covenant public key being incorrectly mutated to a wrong public key.

Since the proposer and approver use the same public key decoding function. This MFA request will be approved, and the wrong covenant public keys will be used. The result is that during unbonding, covenants using public keys with the 0x03 format cannot sign the request and might not be able to reach quorum, which can lead to BTC getting stuck.

Recommendation: Change `parseCovenantKeysFromSlice()` to directly call `btcec.ParsePubKey`:

```diff
-     if len(fpPkBytes) == 33 && (fpPkBytes[0] == 0x02 || fpPkBytes[0] == 0x03) {
-         fpPkBytes = fpPkBytes[1:]
-     }
-     fpSchnorrKey, err := schnorr.ParsePubKey(fpPkBytes)
+     covenantPublicKey, err := btcec.ParsePubKey(fpPkBytes)
+
      if err != nil {
          return nil, err
      }
-     covenantPubKeys[i] = fpSchnorrKey
+     covenantPubKeys[i] = covenantPublicKey
  }

  return covenantPubKeys, nil
```

Lombard: Fixed in PR 70.

Cantina Managed: Fix looks good.

3.3.10 Incomplete validation of the Babylon deposit can lead to Babylon dismissing the BTC transaction or indefinite lockup of funds

Severity: Medium Risk

Context: babylon_deposit.go#L36-L76

Description: As part of the Babylon deposit MFA request, `validateBabylonDepositRequest()` validates `BabylonStakingDeposit`. While most properties are properly validated, some are not validated at all or not verified appropriately:

1. `BabylonStakingDeposit.TxnLockHeight`: Specifies the lock height (in blocks) at which the BTC transaction can be included (mined) in a block. This property is not validated, which could lead to a deposit request being approved, where the BTC transaction might only get mined far in the future. As a result, the BTC funds would be locked, resulting in liquidity issues for Lombard or, worse, a loss of funds. Therefore, it should be ensured that `TxnLockHeight` is not set.

2. `BabylonStakingDeposit.Value`: Corresponds to the staking amount in BTC. While this value is potentially validated and restricted via Cubist policies (`min_staking_value`, `max_staking_value`), it is not verified that the sum of the transaction input values equals the `Value`. Any delta would be considered miner fees, which could lead to a loss of funds for Lombard. Furthermore, it should be validated that the `Value` is within Babylon's bounds of `BabylonStakingParams.MinStakingAmount` and `BabylonStakingParams.MaxStakingAmount`.

3. `BabylonStakingDeposit.LockTime`: Specifies the lock time used for the withdrawal output in the staking deposit transaction. Currently, it is validated by `validateLockTime()` (common.go#L240-L250) to be within the bounds of `[1, math.MaxUint16]`. However, it fails to validate that the `LockTime` is within Babylon's bounds of `BabylonStakingParams.MinStakingTime` and `BabylonStakingParams.MaxStakingTime`.

It is important to ensure that the deposit request is sufficiently validated, as otherwise, Babylon's indexer might ignore the BTC transaction, which would result in Lombard having to wait until the timelock expires to access the BTC funds again. Specifically, Babylon's indexer validates a staking transaction with `validateStakingTx()`(indexer.go#L1056-L1083) and `isOverflow()` (indexer.go#L1085C27-L1107). However, the validations in the latter function are not possible to be replicated reliably by Lombard's approver as data such as Babylon's TVL might change between the time of the deposit request and the time when the transaction is mined and validated by Babylon. Additionally, this information is not exposed anyway by the Babylon indexer nor their Staking API.

Recommendation: Consider validating the `BabylonStakingDeposit` properties as mentioned above to ensure Lombard's Babylon deposit transactions are successfully processed by the indexer. In a similar fashion, it is recommended to add these validations also to the early unbond and timelock withdrawal Babylon requests.

Lombard: Confirmed and fixed in PR 69, PR 84 and commit 35457f96. For Babylon deposit, we definitely need to check if the remaining fee (total inputs in the PSBT minus the request's `value` to be staked) covers the miner fee. Added a fix in PR 89.

Cantina Managed: Fixes look good.

3.4 Low Risk

3.4.1 Possible to approve babylon request with invalid finality provider public key

Severity: Low Risk

Context: finality_providers.go#L12-L24

Description: Possible to approve babylon request with invalid finality provider public key.

Whenever a Babylon request is processed it will go throught a series of validations, one of them being `validateFinalityProviderPks` (common.go#L252-L271).

Such validation of Babylon's request happen in the following functions:

- babylon_deposit.go#L62: `validateBabylonDepositRequest` (staking).
- babylon_early_unbond.go#L50: `validateBabylonEarlyUnbondRequest` (unbond).
- babylon_timelock_withdraw.go#L54: `validateBabylonTimelockWithdrawRequest` (withdraw).

```go
func (c *cubistCustodianManager) validateFinalityProviderPks(finalityProviderPk string) error {
    var isWhitelistedFpPk bool

    whitelistedFpPks, err := c.babylonService.GetFinalityProviderPks()
    if err != nil {
        return errors.Wrap(err, "error getting Babylon finality providers")
    }

    for _, pk := range whitelistedFpPks {
        if strings.EqualFold(pk, finalityProviderPk) {
            isWhitelistedFpPk = true
        }
    }

    if !isWhitelistedFpPk {
        return errors.Errorf("`finality_provider_pk` (%s) is not whitelisted", finalityProviderPk)
    }

    return nil
}
```

1. We can observe that it will reach the Babylon staking API to confirm that the finality provider supported by the approver are actually valid and as such returning an array of supported key.
2. It will then try to confirm (`isWhitelistedFpPk = true`) if the request's finality provider key is present in that returned array.

The problematic situation originate from the babylon internal service `GetFinalityProviderPks` (service.go#L61-L73) which lacks proper validation for a specific case. Whenever a key from the config doesn't exist (so not found by the Babyblon Staking API), it will still be counted as a valid key, as this will not return an error, which is unexepected, and will translate into the approver service to accept request which might not have valid finality provider key.

```go
func (s *service) GetFinalityProviderPks() ([]string, error) {
    res := make([]string, 0)
    for _, fpPk := range s.config.FinalityProviderPks {
        _, err := s.client.FinalityProviders(fpPk)
        if err != nil {
            return nil, err
        }

        res = append(res, fpPk)
    }

    return res, nil
}
```

Proof of Concept: Create a file `internal/babylon/client/finality_providers_test.go` and paste the following content. All tests will pass, except `key well formed but not found` confirming this report since not returning an error as it should.

```go
package client

import (
    "io"
    "net/http"
    "net/http/httptest"
    "net/url"
    "testing"
    "time"

    v1 "github.com/lombard-finance/approver/internal/babylon/api/v1"
    "github.com/sirupsen/logrus"
    "github.com/stretchr/testify/assert"
)

func TestFinalityProviders(t *testing.T) {
    // Test cases
    tests := []struct {
        name         string
        fpPk         string
        serverStatus int
        serverResp   string
        expectedResp *v1.FinalityProvidersResponse
        expectError  bool
    }{
        {
            name:         "successful_response - single",
            fpPk:         "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
            serverStatus: http.StatusOK,
            serverResp: `{
                "data": [{
                    "description": {
                        "moniker": "test-fp",
                        "identity": "test-identity",
                        "website": "test.com",
                        "security_contact": "security@test.com",
                        "details": "test details"
                    },
                    "commission": "0.1",
                    "btc_pk": "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
                    "active_tvl": 1000000,
                    "total_tvl": 2000000,
                    "active_delegations": 10,
                    "total_delegations": 20
                }]
            }`,
            expectedResp: &v1.FinalityProvidersResponse{
                Data: []v1.FinalityProvider{
                    {
                        Description: v1.FinalityProviderDescription{
                            Moniker:         "test-fp",
                            Identity:        "test-identity",
                            Website:         "test.com",
                            SecurityContact: "security@test.com",
                            Details:         "test details",
                        },
                        Commission:        "0.1",
                        BtcPk:             "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611
",
                        ActiveTVL:         1000000,
                        TotalTVL:          2000000,
                        ActiveDelegations: 10,
                        TotalDelegations:  20,
                    },
                },
            },
            expectError: false,
        },
        {
            name:         "successful_response - multiple",
            fpPk:         "",
            serverStatus: http.StatusOK,
            serverResp: `{
                "data": [{
                    "description": {
```

```
                    "moniker": "test-fp",
                    "identity": "test-identity",
                    "website": "test.com",
                    "security_contact": "security@test.com",
                    "details": "test details"
                },
                "commission": "0.1",
                "btc_pk": "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
                "active_tvl": 1000000,
                "total_tvl": 2000000,
                "active_delegations": 10,
                "total_delegations": 20
            },
            {
                "description": {
                    "moniker": "test-fp",
                    "identity": "test-identity",
                    "website": "test.com",
                    "security_contact": "security@test.com",
                    "details": "test details"
                },
                "commission": "0.1",
                "btc_pk": "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
                "active_tvl": 1000000,
                "total_tvl": 2000000,
                "active_delegations": 10,
                "total_delegations": 20
            }]
        }`,
        expectedResp: &v1.FinalityProvidersResponse{
            Data: []v1.FinalityProvider{
                {
                    Description: v1.FinalityProviderDescription{
                        Moniker:         "test-fp",
                        Identity:        "test-identity",
                        Website:         "test.com",
                        SecurityContact: "security@test.com",
                        Details:         "test details",
                    },
                    Commission:        "0.1",
                    BtcPk:             "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611
",
                    ActiveTVL:         1000000,
                    TotalTVL:          2000000,
                    ActiveDelegations: 10,
                    TotalDelegations:  20,
                },
                {
                    Description: v1.FinalityProviderDescription{
                        Moniker:         "test-fp",
                        Identity:        "test-identity",
                        Website:         "test.com",
                        SecurityContact: "security@test.com",
                        Details:         "test details",
                    },
                    Commission:        "0.1",
                    BtcPk:             "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611
",
                    ActiveTVL:         1000000,
                    TotalTVL:          2000000,
                    ActiveDelegations: 10,
                    TotalDelegations:  20,
                },
            },
        },
        expectError: false,
    },
    {
        name:         "server_error",
        fpPk:         "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
        serverStatus: http.StatusInternalServerError,
        serverResp:   `{"error": "internal server error"}`,
        expectedResp: nil,
        expectError:  true,
    },
    {
            name:         "invalid_json_response",
            fpPk:         "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
            serverStatus: http.StatusOK,
            serverResp:   `{invalid json}`,
            expectedResp: nil,
            expectError:  true,
        },
        {
            //curl https://staking-api.staging.babylonchain.io/v1/finality-providers?fp_btc_pk=f712efa8037fa13
↪   cf57388ab32731ff706f3028142b3800e132f77b02bab1012
            name:         "key well formed but not found",
            fpPk:         "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
            serverStatus: http.StatusOK,
            serverResp:   `{"data":null}`,
            expectedResp: nil,
            expectError:  true,
        },
        {
            //curl https://staking-api.staging.babylonchain.io/v1/finality-providers?fp_btc_pk=f712efa8037fa13
↪   cf57388ab32731ff706f3028142b3800e132f77b02bab101
            name:         "malformed key",
            fpPk:         "02c7e138bff86a4cc33e6871ce9c907482795653e58a30305400c0da3226a6d611",
            serverStatus: http.StatusBadRequest,
            serverResp:   `{"errorCode":"BAD_REQUEST","message":"invalid fp_btc_pk"}`,
            expectedResp: nil,
            expectError:  true,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // Create test server
            server := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                // Verify request method
                assert.Equal(t, http.MethodGet, r.Method)

                // Verify path and query parameters
                assert.Contains(t, r.URL.Path, "/v1/finality-providers")
                assert.Equal(t, tt.fpPk, r.URL.Query().Get("fp_btc_pk"))

                // Set response
                w.WriteHeader(tt.serverStatus)
                w.Header().Set("Content-Type", "application/json")
                _, err := io.WriteString(w, tt.serverResp)
                assert.NoError(t, err)
            }))
            defer server.Close()

            // Create client
            serverURL, err := url.Parse(server.URL)
            assert.NoError(t, err)

            client := &Client{
                logger:  logrus.NewEntry(logrus.New()),
                base:    serverURL,
                client:  server.Client(),
                timeout: 30 * time.Second,
            }

            // Call FinalityProviders
            resp, err := client.FinalityProviders(tt.fpPk)

            if tt.expectError {
                assert.Error(t, err)
                assert.Nil(t, resp)
            } else {
                assert.NoError(t, err)
                assert.Equal(t, tt.expectedResp, resp)
            }
        })
    }
}
```

Recommendation: When the finality provider key is not found, that should return an error, which would prevent to approve request with not real finality provider key:

```
  func (cli *Client) FinalityProviders(fpPk string) (*v1.FinalityProvidersResponse, error) {
      response, err := cli.get(fmt.Sprintf("/v1/finality-providers?fp_btc_pk=%s", url.PathEscape(fpPk)))
      if err != nil {
          return nil, errors.Wrap(err, "request GetFinalityProviders")
      }

      decoded, err := utils.DecodeJSONResponse[v1.FinalityProvidersResponse](response)
      if err != nil {
          return nil, errors.Wrap(err, "decode FinalityProvidersResponse")
      }
+
+     if len(decoded.Data) == 0 {
+         return nil, errors.New("Key not found!")
+     }
+
      return &decoded, nil
  }
```

Lombard: Fixed in PR 68.

Cantina Managed: Fix looks good.

3.4.2 Addressbook service loads config with incorrect namespace and defaults to default values

Severity: Low Risk

Context: config.go#L20-L26

Description: `ParseFromViper()` loads the config for the `addressbook` service and defaults to the provided default value if no matching key is found in the config file. According to the example config file, `config-example.yaml`, the namespace is supposed to be `addresses`.

```yaml
addresses:
  url: "https://bft-devnet.stage.lombard.finance"
  timeout: "30s"
```

However, `ParseFromViper()` uses the namespace `addressbook` for the config keys. As a result, the config file is not being loaded correctly, and the default values are being used instead.

Recommendation: Consider either updating the config file to use the `addressbook` namespace or updating the `ParseFromViper()` function to use the correct namespace `addresses`.

Lombard: Fixed in commit 01a62869 on `audit/veridise` branch along with some other issues that were identified during QA.

Cantina Managed: Fix looks good.

3.4.3 Outages in external APIs can lead to legitimate MFA requests being rejected

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The Lombard approver depends on making HTTP requests to two external APIs, the Babylon API and the Address Book API. Thus it is possible, that these services experience temporary outages which will affect the approver in the sense that it will reject the legitimate MFA request since an error is returned (due to timeouts).

- internal/babylon/client/finality_providers.go#L13-L16:

```go
response, err := cli.get(fmt.Sprintf("/v1/finality-providers?fp_btc_pk=%s", url.PathEscape(fpPk)))
if err != nil {
    return nil, errors.Wrap(err, "request GetFinalityProviders")
}
```

- internal/babylon/client/versions.go#L11-L14:

```
response, err := cli.get("/v1/global-params")
if err != nil {
    return nil, errors.Wrap(err, "request GetGlobalParams")
}
```

- internal/addressbook/address.go#L16-L19:

```
response, err := s.get(fmt.Sprintf("/api/v1/address/%s", req.GetBtcAddress()))
if err != nil {
    return nil, errors.Wrap(err, "request GetAddress")
}
```

Recommendation Consider retrying the HTTP request on network errors.

Lombard: Fixed in PR 59.

Cantina Managed: Fix looks good. 2 questions:

- There is an inconsistency in the status code, retry condition (> 500) vs what you consider success in the `get` function (2XX), not sure if this is expected. The question really is, when the external service is down, will it always endup in a status code > 500 for the client, maybe worth asking:

```
AddRetryCondition(
    func(r *resty.Response, err error) bool {
        return err != nil || r.StatusCode() >= http.StatusInternalServerError
    },
).
```

- Is `defer resp.Body` really not needed anymore?

Lombard:

1. Added a retry condition for 429's in PR 83. Based on the Babylon docs and the code we have for our address book service, it seems sufficient to just retry for 429's and >= 500s.
2. It's not needed anymore, but I didn't really like how the code was written, so I changed this a bit in the PR mentioned in this comment.

Cantina Managed: Fixes look good.

3.4.4 Missing validating PSBT input `WitnessUtxo.PkScript` against the actual UTXO `ScriptPubKey` allows bypassing address validation

Severity: Low Risk

Context: internal/approver/strategy/cubist/common.go#L139-L140, internal/approver/strategy/cubist/common.go#L
L109,

Description: In `validateStakingInputs()` and `validateDepositInputs()`, the PSBT's `Input` are enumerated, and the address is extracted from the input's witness script via `txscript.ExtractPkScriptAddrs()`. The extracted address is then subsequently validated to ensure it is a valid and expected address.

However, the PSBT inputs are just metadata, they are not guaranteed to match the actual transaction (`UnsignedTx`). Therefore, it is necessary to double-check the inputs against the Bitcoin network to ensure they are legitimate UTXOs.

While this is done in `validateStakingInputs()` and `validateStakingInput()` by calling `c.blockchainService.GetTxOut(&outPoint)` to retrieve and validate the UTXO's value, it is missing to validate `in.WitnessUtxo.PkScript` against the actual UTXO's `ScriptPubKey` in `validateDepositInputs()` and `validateStakingInputs()`.

As a result, it is possible to bypass the address validation by using a different `WitnessUtxo.PkScript` in the PSBT's `Input` than the actual UTXO's `ScriptPubKey`.

Recommendation: Consider validating that the `WitnessUtxo.PkScript` matches the actual UTXO's `ScriptPubKey` to ensure the address validation is accurate.

Lombard: Fixed in PR 61.

Cantina Managed: Fix looks good.

3.4.5 Lombard transfer can include UTXOs with very small amounts of BTC that make them economically unspendable

Severity: Low Risk

Context: common.go#L166-L203

Description: Lombard transfer MFA requests are validated by the approver in `validateLombardTransferRequest()`. The transaction outputs are validated by `validateStakingOutputs()`, which ensures that the address of the outputs matches the Cubist staking addresses.

However, it misses out on validating the output values, which would allow approving a transaction that contains a lot of UTXOs with very small amounts of BTC (i.e., "dust"). Such UTXOs are likely not worth spending in a transaction as the transaction fees would be higher than the combined value of the UTXOs, thus, making them unspendable.

Recommendation: Consider validating that the value of an output is greater than a certain minimum threshold to prevent the inclusion of dust UTXOs in the transaction. It is observed that such a threshold (`MinOutputsAmount`) is already present but unused in internal/approver/strategy/constants.go#L12.

Similarly, it can be considered to also enforce an upper cap on the output value, by using `MaxOutputsAmount`, which is currently also unused and set to `int64(btcutil.SatoshiPerBitcoin * 10)`, i.e. 10 BTC.

Lombard: Fixed in PR 74.

Cantina Managed: Fix looks good.

3.4.6 Fee depletion via many useless Bitcoin transactions

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: A malicious proposer can choose to grief by sending many Bitcoin transactions which are ultimately useless. For instance they can send a Bitcoin transaction that assigns only the change UTXO as the output. This will cause the approver to skip any outputs processing and skip to validating the change UTXO.

This will be considered a useless transaction, because they will only be sending the UTXO back to themselves. For example, it is possible to abuse this in `lombard_unstake` when the miner fee issue is resolved to slowly deplete Bitcoin the staking wallet over time.

Recommendation: Verify that there is at least one useful output UTXO (other than the change UTXO) in the Bitcoin transaction, this will ensure some useful work is done commensurate with the BTC fee paid from the BTC staking wallet.

Lombard: Fixed in PR 73 and PR 78.

Cantina Managed: Fix looks good.

3.4.7 Missing check that the deposit address is not a multisig

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: There is a missing check that the deposit address is not a multisig in `validateDepositInputs`. As we can see, we only ensure that the `addrs` is not equal to 0 before proceeding, which allows multisig deposit address to be used

- internal/approver/strategy/cubist/common.go#L138-L147:

```
// Extract address from input's pubkey script
_, addrs, _, err := txscript.ExtractPkScriptAddrs(in.WitnessUtxo.PkScript,
    c.blockchainService.GetNetwork())
if err != nil {
    return errors.Wrapf(err, "error extracting addresses from pubkey script for PSBT input (%d)", i)
}

if len(addrs) == 0 {
    return errors.Errorf("no addresses extracted from pubkey script for PSBT input (%d)", i)
}
```

Recommendation: Return an error if `len(addrs) != 1` or `len(numAddrs) != 1`.

Lombard: Fixed in PR 66. Also affects "Invalid public keys can cause the approver service to panic and crash".

Cantina Managed: Fix looks good.

3.4.8 Lombard unstake race condition allowing repeated payouts

Severity: Low Risk

Context: internal/approver/strategy/cubist/lombard_unstake.go#L67-L112

Description: Lombard unstake MFA requests are processed in `executeLombardUnstake()`, which verifies that the transaction outputs match a corresponding unpaid unstake request. This prevents unstake requests from being paid out repeatedly. However, given the asynchronous nature of the overall process, the approver is not immediately aware of the resulting BTC transaction due to the notarization of the unstake receipt taking some time. Consequently, the approver does not mark the unstake request as paid in the database until the receipt is received from the ledger.

Until then, another MFA request for the same unstake will be matched with the same unstake request, which is technically already paid, resulting in repeated payouts.

Recommendation: Consider adding an additional field to the unstake database model (e.g., a timestamp) that allows marking an unstake as tentatively paid, which can be immediately set upon approving the unstake MFA request. This way, race conditions can be mitigated. Additionally, should the notarized unstake receipt not be received within a reasonable time frame, the unstake entry in the database can be "unlocked" again to allow for a new unstake request to be matched with it and paid out.

Lombard: Fixed in PR 82.

Cantina Managed: It might make sense to call `BulkApproveUnstakes()` before approving the MFA request. This would allow handling an error that occurs when bulk updating the unstakes and only approving the MFA if everything succeeded. Otherwise, you might end up with an approved MFA request without updating the `approved_at` timestamp in the database.

Also should be approving only matched unstakes, not all unpaid unstakes.

Lombard:

- Update db before approving on Cubist; see commit a96b8453.
- Only approve matched unstakes; see commit 5c5c967e.

Cantina Managed: Fixes look good.

3.4.9 Potential data inconsistency caused by errors during unstake receipt processing

Severity: Low Risk

Context: unstake.go#L17

Description: In `processUnstakeNotarizations()`, there is an edge case for new approver instances that try to sync up with Lombard's ledger, leading to data inconsistencies. In the worst case, it can lead to potential double payouts for unstake requests if a majority of approvers are affected.

The issue occurs because unstakes are processed and atomically stored in the database in bulk via `BulkCreateUnstakes()` while receipts are handled sequentially. If processing a receipt errors in `processReceipt()`, the failed receipt is skipped while the remaining receipts are processed.

However, it subsequently checks whether all receipts are successfully processed (unstake.go#L104-L106), and if not, returns an error:

```
if len(receipts)-receiptSucessCount > 0 {
    return errors.Errorf("failed to process %d unstake receipts", len(receipts)-receiptSucessCount)
}
```

This error propagates up to the `processUnstakeNotarizations()` caller and early exits the `for` loop (unstake.go#L34).

At the same time, due to storing the unstakes in bulk *before* the receipts are processed, the oldest pending ID, which is used as a pagination offset when querying notary sessions from the ledger, is advanced in a way that the failed receipts will be skipped.

For example, consider a new approver instance that tries to sync up with the ledger. The following sequence of unstakes and unstake receipts is processed:

- Unstakes: `[0, 1, 2, 3, 4, 10, 11, 12]`
- Receipts: `[5, 6, 7, 8, 9, 13, 14, 15]`

Note: All unstakes (except ID 12) and receipts are fully notarized, i.e., their status is `Completed`. Unstake 12 is submitted to the ledger but not yet fully notarized, i.e., its status is `Pending`.

Processing sequence:

1. `BulkCreateUnstakes()` atomically stores unstakes 0-4 and 10-12 in bulk in the database. The oldest pending ID is now 12.
2. Receipts are sequentially processed by `processReceipt()`: a. Receipt 5 successfully processed and stored in the database. b. Receipt 6 fails to process. c. Receipts 7-15 are successfully processed and stored in the database. The oldest pending ID is now 15.
3. As not all receipts are successfully processed, the error `"failed to process one unstake receipt"` is returned $\rightarrow$ `for` loop in `processUnstakeNotarizations()` early exits.
4. The next time `processUnstakeNotarizations()` is called, it will query notary sessions from the ledger starting from the oldest pending ID, which is 15.

As a result, the receipt with ID 6 will not be reprocessed and thus fails to be stored in the database. As this receipt is associated with the unstake ID 1, the unstake will not be marked as paid and thus used when approving Lombard unstake MFAs to match transaction outputs (lombard_unstake.go#L83-L99).

Recommendation: Consider explicitly tracking the last successfully processed unstake or receipt ID to be able to resume processing from the correct offset in case of an error.

Lombard: Fixed in PR 81 along "All notary sessions starting from offset 0 are repeatedly processed when there are no pending unstakes or unstake receipts" since both findings are somewhat related.

Cantina Managed: Fix looks good.

3.5 Gas Optimization

3.5.1 Performance optimization when processing unstake notary sessions

Severity: Gas Optimization

Context: unstake.go#L51-L72

Description: Minor performance optimization when processing unstake notary sessions as unstake sessions are mutually exclusive (`unstake` vs `unstake receipt`).

Recommendation: Simply put the unstake receipt processing into an `else`:

```
  if bytes.Equal(payloadSelector, notaryexported.UnstakeSelector) {
      unstake, err := ledger.DecodeUnstake(ns)
      if err != nil {
          s.logger.WithError(err).Error("error decoding unstake notarization")
          continue
      }

      unstakes = append(unstakes, unstake)
- }
-
- if bytes.Equal(payloadSelector, notaryexported.UnstakeExecutionSelector) {
+ } else if bytes.Equal(payloadSelector, notaryexported.UnstakeExecutionSelector) {
      receipt, err := ledger.DecodeUnstakeReceipt(ns)
      if err != nil {
          s.logger.WithError(err).Error("error decoding unstake receipt")
          continue
      }

      receipts = append(receipts, receipt)
  }
```

Lombard: Fixed in PR 47, specifically commit 1e40095d.

Cantina Managed: Fix looks good.

3.5.2 Redundant condition in `BulkCreateUnstakes`

Severity: Gas Optimization

Context: unstake.go#L115-L116

Description: Redundant condition in `BulkCreateUnstakes`. The `for/range` is taking care of skipping in case the array length is zero, which make this condition redundant.

Recommendation: Remove the condition.

```
  // Update unstake states
- if len(toUpdate) > 0 {
      for _, unstake := range toUpdate {
          if err := tx.Model(&model.Unstake{}).
              Where("id = ?", unstake.ID).
              Update("state", strings.ToLower(unstake.State)).Error; err != nil {
              return errors.Wrap(err, "error updating unstake state")
          }
      }
- }
```

Lombard: Fixed in PR 48, specifically commit 7e2221a8.

Cantina Managed: Fix looks good.

3.5.3 Redundant condition in `processMfaRequest`

Severity: Gas Optimization

Context: custodian_manager.go#L95

Description: Redundant condition in `processMfaRequest`. The first part of the condition is redundant. Proven here:

(See this code in the go playground here)

```go
package main

import "fmt"

type StructA struct {
    Receipt *StructB
}

type StructB struct {
    Confirmation string
}

func (o *StructA) GetReceipt() *StructB {
    if o == nil || o.Receipt == nil {
        return nil
    }
    return o.Receipt
}

func (o *StructB) GetConfirmation() string {
    if o == nil || o.Confirmation == "" {
        var ret string
        return ret
    }
    return o.Confirmation
}

func main() {
    var structa StructA
    if structa.GetReceipt() != nil && structa.GetReceipt().GetConfirmation() != "" {
        fmt.Println("Confirmation is good")
    } else {
        fmt.Println("No confirmation (nil case - 2 checks)")
    }

    if structa.GetReceipt().GetConfirmation() != "" {
        fmt.Println("Confirmation is good")
    } else {
        fmt.Println("No confirmation (nil case - 1 check)")
    }

    structaReal := &StructA{Receipt: &StructB{Confirmation: "allo"}}
    if structaReal.GetReceipt().GetConfirmation() != "" {
        fmt.Printf("Confirmation is good: %s\n", structaReal.GetReceipt().GetConfirmation())
    } else {
        fmt.Println("No confirmation (nil case - 1 check)")
    }
}
```

Recommendation: Remove the redundant condition.

```diff
  func (c *cubistCustodianManager) processMfaRequest(ctx context.Context, mfaReq *cubistv0.MfaRequestInfo)
↪   error {
-     if mfaReq.GetReceipt() != nil && mfaReq.GetReceipt().GetConfirmation() != "" {
+     if mfaReq.GetReceipt().GetConfirmation() != "" {
          c.logger.Info("NFA request is already confirmed", "id", mfaReq.GetId())
          return nil
      }
  // ...
```

Lombard: Fixed in PR 52, specifically commit 5801c0c1.

Cantina Managed: Fix looks good.

3.5.4 Excessive resource consumption in `validateUnstakingOutputs`

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: In `validateUnstakingOutputs`, we iterate over every output in the Bitcoin transaction:

- internal/approver/strategy/cubist/lombard_unstake.go#L92-L99:

```go
// Ignore the last output because it's the change output
for i, out := range outputs[:len(outputs)-1] {
    err := c.validateUnstakingOutput(ctx, out, unpaidUnstakes, matchedUnstakes)
    if err != nil {
        return errors.Wrapf(err, "error validating unstaking output %d", i)
    }

    unstakeTotal += out.Value
}
```

In each iteration, we iterate over every unpaid unstake fetched from the database:

- internal/approver/strategy/cubist/lombard_unstake.go#L120-L137:

```go
for _, unpaidUnstake := range unpaidUnstakes {
    // Skip already matched unstakes
    if _, exists := matchedUnstakes[unpaidUnstake.ID]; exists {
        continue
    }

    if int64(unpaidUnstake.Amount) == out.Value && bytes.Equal(unpaidUnstake.ScriptPubKey,
↪   out.PkScript) {
        // Validate the unstake again against the Ledger
        if err := c.validateUnstakeOnLedger(ctx, out, unpaidUnstake); err != nil {
            c.logger.WithError(err).Errorf("expected output to match unstake (%d) on Ledger, skipping",
↪   unpaidUnstake.ID)
            continue
        }

        // Mark as matched and return
        matchedUnstakes[unpaidUnstake.ID] = struct{}{}
        return nil
    }
}
```

The time complexity of `validateUnstakingOutputs` is thus $\mathcal{O}(\mathtt{mn})$ where `m` is the number of outputs and `n` is the number of unpaid unstakes. If there are too many outputs or unpaid unstakes, resource consumption can become too excessive.

Recommendation: We believe that this can be potentially refactored to a time complexity of $\mathcal{O}(\mathtt{m}+\mathtt{n})$ by using a map to store `(amount, ScriptPubkey)` pair and a counter variable to calculate how many outputs are successfully matched. Then iterate through the unpaid unstakes to find a matching output. If not all outputs are successfully matched then return an error. Although, it must be checked thoroughly that this doesn't change any crucial logic in the code.

Lombard: Fixed in PR 80.

Cantina Managed: Fix looks good. Not sure we need the `matchedUnstakes` map now since all the unstakes we will be iterating over have unique IDs. We can probably convert it to an `uint64` and check the count. But the way it is now is also acceptable. It is up to the client.

Lombard: We think it's still necessary to keep `matchedUnstakes`.

3.5.5 Babylon request are unmarshalled twice during their processing

Severity: Gas Optimization

Context: common.go#L362-L380, custodian_manager.go#L139-L154

Description: Babylon request are always unmarshalled twice during their processing which is inefficient.

Balylon MFA requests are unmarshalled in two places during their processing life cycle:

1. `processMfaRequest` (custodian_manager.go#L139-L154) when starting processing the request.
2. In `executeBabylon*` functions, `getBabylon*` functions are called, which calls `getBabylonStakingRequest` (common.go#L362-L380) which is redoing what was already done in `processMfaRequest`, which is just a waste.

Recommendation: Refactor the code to only unmarshal once per request (you could pass down the unmarshalled struct in `executeBabylon*`). It this is somehow by design, please ignore this report.

Lombard: Fixed in PR 60.

Cantina Managed: Fix looks good.

3.6 Informational

3.6.1 Risk of updating a unstake for nothing in `BulkCreateUnstakes`

Severity: Informational

Context: unstake.go#L97

Description: Risk of updating an unstake for nothing in `BulkCreateUnstakes`. The string comparison can be problematic and error-prone in the moment, it requires only one instance to forget to sanitize the input to create a bug.

Recommendation: Use `strings.EqualFold` instead of raw string comparaison:

```
  // Check if state needs updating
- if existingUnstake.State != unstake.State {
+ if !strings.EqualFold(existingUnstake.State, unstake.State) {
      toUpdate = append(toUpdate, unstake)
      updated++
      continue
  }
```

Lombard: Fixed in PR 49, specifically commit 76747503.

Cantina Managed: Fix looks good.

3.6.2 Addressbook hiding not found BTC address behind a decoding error

Severity: Informational

Context: service.go#L92-L96

Description: Addressbook hiding not found BTC address behind a decoding error. There is a minor issue in the `addressbook` service. For a specific case, whenever the requested BTC address is not found (service.go#L92-L96), it doesn't return any error and with a nil response.

Nevertheless, `DecodeJSONResponse` (address.go#L21-L24) will return an error as `body is nil`:

```go
func DecodeJSONResponse[T any](body io.Reader) (T, error) {
    var res T

    if body == nil {
        return res, errors.New("no body to read")
    }

    if err := json.NewDecoder(body).Decode(&res); err != nil {
        return res, errors.Wrap(err, "decode body")
    }

    return res, nil
}
```

So, in essence, this will make the only client (common.go#L149-L155) of `GetAddress` to still return an error, which I don't think is unexpected, but the distinction between not found and real error is lost.

Proof of Concept: Create a file `internal/addressbook/service_test.go` and paste the following content. All tests will pass, confirming this report ("not found" → error)

```go
package addressbook

import (
    "context"
    "encoding/json"
    "net/http"
    "net/http/httptest"
    "net/url"
    "testing"
    "time"

    "github.com/lombard-finance/go-common/config"
    pbaddressbook "github.com/lombard-finance/proto-contract/pkg/addressbook"
    "github.com/sirupsen/logrus"
    "github.com/spf13/viper"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

type testConfigProvider struct {
    v *viper.Viper
}

func newTestConfigProvider(URL string, timeout time.Duration) *testConfigProvider {
    v := viper.New()

    v.Set("addressbook.url", URL)
    v.Set("addressbook.timeout", timeout)

    return &testConfigProvider{v: v}
}

func (p *testConfigProvider) Parse(cfg config.IConfig) error {
    return cfg.ParseFromViper(p.v)
}

func TestGetAddress(t *testing.T) {
    tests := []struct {
        name              string
        btcAddress        string
        serverResponse    *pbaddressbook.GetAddressResponse
        rawServerResponse string
        serverStatus      int
        expectedError     string
    }{
        {
            name:       "successful request",
            btcAddress: "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh",
            serverResponse: &pbaddressbook.GetAddressResponse{
                Address: &pbaddressbook.Address{
                    BtcAddress: "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh",
                },
            },
            serverStatus: http.StatusOK,
        },
        {
            name:          "empty address",
            btcAddress:    "",
            serverStatus:  http.StatusOK,
            expectedError: "`address` cannot be empty",
        },
        {
            name:          "server error",
            btcAddress:    "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh",
            serverStatus:  http.StatusInternalServerError,
            expectedError: "Method GET, StatusCode: 500",
        },
        {
            name:              "not found",
            btcAddress:        "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh",
            serverStatus:      http.StatusNotFound,
            rawServerResponse: "NOT FOUND",
            expectedError:     "no body to read",
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // Create test server
            server := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                // Verify request method
                assert.Equal(t, http.MethodGet, r.Method)

                // Check path contains the BTC address
                expectedPath := "/api/v1/address/" + tt.btcAddress
                assert.Equal(t, expectedPath, r.URL.Path)

                // Set response status
                w.WriteHeader(tt.serverStatus)

                // Write response if we have one
                if tt.serverResponse != nil {
                    json.NewEncoder(w).Encode(tt.serverResponse)
                }

                // Write response if we have one
                if tt.rawServerResponse != "" {
                    w.Write([]byte(tt.rawServerResponse))
                }
            }))
            defer server.Close()

            // Parse server URL
            serverURL, err := url.Parse(server.URL)
            require.NoError(t, err)
            require.NotNil(t, serverURL)

            // Create serviceAB with mock config
            logger := logrus.New()
            serviceAB := NewService(logger)

            // Start serviceAB
            err = serviceAB.Start(context.Background(), newTestConfigProvider(server.URL, time.Minute))
            require.NoError(t, err)
            defer serviceAB.Stop()

            // Make request
            req := &pbaddressbook.GetAddressRequest{
                BtcAddress: tt.btcAddress,
            }
            resp, err := serviceAB.GetAddress(req)

            // Check error
            if tt.expectedError != "" {
                require.Error(t, err)
                assert.Contains(t, err.Error(), tt.expectedError)
                return
            }
            // Check successful response
            require.NoError(t, err)
            require.NotNil(t, resp)
            assert.Equal(t, tt.serverResponse.Address.BtcAddress, resp.Address.BtcAddress)
        })
    }
}
```

Recommendation: Since it will return an error anyway in `GetAddress`, consider making the following changes so that the intended behavior is more clear, and the error is not hidden behind a `decode body error`. Also, there is no point in trying to decode if the response is `nil`:

```
  if resp.StatusCode == http.StatusNotFound {
      // Nothing found.  This is not an error, so we return nil on both counts.
+      log.Warn("request successful, but address not found!")
      cancel()
-      return nil, nil
+      return nil, errors.New("address not found!")
  }
```

Lombard: Fixed in PR 50. The recommendation is to log and return an error in the `AddressBook` client's `get` function if this address isn't found. Even though there's currently only 1 API (`GetAddress`) that calls it, we think it's too easy to forget to change this if we add any new GET APIs in AddressBook service. So the fix we applied is specifically in the `GetAddress` function

Cantina Managed: Fix looks good.

3.6.3 Increase consistency in string comparison using `EqualFold`

Severity: Informational

Context: mfa.go#L22

Description: More consistency in string comparison.

Recommendation: Use `strings.EqualFold` to be more consistent across the code base, like in `IsApprovedByUser()` (mfa.go#L12):

```
  func IsAllowedToApprove(mfaReq *cubistv0.MfaRequestInfo, userId string) bool {
      for _, approver := range mfaReq.GetStatus().AllowedApprovers {
-         if strings.ToLower(approver) == strings.ToLower(userId) {
+         if strings.EqualFold(approver, userId) {
              return true
          }
      }

      return false
  }
```

Lombard: Fixed in PR 53, specifically commit e4565c8a.

Cantina Managed: Fix looks good.

3.6.4 Typographical errors

Severity: Informational

Context: custodian_manager.go#L96

Description/Recommendation: `NFA` should be changed for `MFA`.

Lombard: Fixed in commit 27daec96be4ad6616a38219a05e5f5156d61524a on `dev` branch along with some other issues that were identified during QA.

Cantina Managed: Fix looks good.

3.6.5 Unused code

Severity: Informational

Context: constants.go#L7-L13, utils.go#L91, psbt.go#L44

Description/Recommendation: The lines of code specified in the context are not used in the codebase. They should be removed.

Lombard: Fixed in PR 63, specifically commit fa753aed.

Cantina Managed: Fix looks good.

3.6.6 Transaction output value will be truncated in logged error message

Severity: Informational

Context: internal/approver/strategy/cubist/common.go#L80, internal/approver/strategy/cubist/common.go#L102

Description: Casting the float value `txOut.Value` to `int64` will truncate the decimals, resulting in a confusing error message that logs a very small transaction output value.

Recommendation: Consider multiplying `txOut.Value` with `1e8`, the same as in the `if` condition above.

Lombard: Fixed in PR 55, specifically commit 9e5f06e9.

Cantina Managed: The instance in L80 is fixed. Please also consider fixing the instance in common.go#L102 as well.

Lombard: We noticed it while working on a separate finding and the fix is already on `audit/spearbit` branch (see common.go#L102).

Cantina Managed: Fix looks good.

3.6.7 Use `.Fatal` instead of `Fatalf` for logging error messages that do not contain placeholders

Severity: Informational

Context: internal/database/gorm.go#L35, internal/database/gorm.go#L42

Description: In two instances, `Fatalf` is used to log an error message. However, `Fatalf` is intended for formatted strings while the error message is not formatted, i.e., it does not contain any placeholders for variables.

Recommendation: Consider using `Fatal` instead of `Fatalf` for logging the error message.

Lombard: Fixed in PR 56, specifically commit 93bfa406.

Cantina Managed: Resolved by updating both instances.

3.6.8 Nil pointer dereference and panic if staking key does not exist

Severity: Informational

Context: common.go#L118-L125, common.go#L192-L199

Description: In `custodian_manager`, we verify that the staking key is not `nil`.

- custodian_manager.go#L200-L210:

```go
inputAddr := addrs[0].EncodeAddress()
stakingKey, err := c.cubistService.GetStakingKey(inputAddr)
if err == nil {
    // verify staking key not nill
    if stakingKey == nil {
        return errors.New("staking key not found")
    }
    // Verify the key is enabled before processing
    if !stakingKey.GetEnabled() {
        return errors.New("staking key is disabled")
    }
```

However, this is not done in the other parts of the codebase:

~~~~
```go
// Validate Cubist staking address
stakingKeyResp, err := c.cubistService.GetStakingKey(addrs[0].EncodeAddress())
if err != nil {
    return errors.Wrapf(err, "error checking on Cubist for the address extracted from pubkey script for PSBT
    ↪  output (%d)", i)
}

if !stakingKeyResp.Enabled {
    return errors.Errorf("staking key is disabled (%s)", stakingKeyResp.MaterialId)
}
```
~~~~

If `stakingKeyResp` returns `nil` due to for example, a non-existent address is provided. The `nil` pointer will be dereferenced in `stakingKeyResp.Enabled`, which will cause the approver to panic and crash.

Recommendation: We recommend using `stakingKeyResp.GetEnabled()` which internally check the `nil` pointer:

```go
// GetEnabled returns the Enabled field value
func (o *GetKeyInOrg200Response) GetEnabled() bool {
    if o == nil {
        var ret bool
        return ret
    }

    return o.Enabled
}
```

Lombard: Fixed in PR 57, specifically commit ed2ff8a8.

Cantina Managed: Fix looks good.

3.6.9 Unstrict decoding of PSBT sign requests

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When decoding the PSBT sign request, it uses `json.Unmarshal` which is by default non-strict JSON decoding:

- internal/approver/strategy/cubist/custodian_manager.go#L162:

```go
if err := json.Unmarshal(bodyJson, &psbtSignReq); err != nil {
```

As such additional fields can be included in the PSBT sign requests which will be ignored and not return and error, which will result in these MFA requests being approved.

Recommendation: Consider using strict JSON unmarshaling via JSON decoder and `DisallowUnknownFields()` which will return an `err` when additional fields are present in the PSBT sign request.

Lombard: Fixed in PR 71. Added some tests and fixes in the follow-up PR 76 since the first one was accidentally merged too soon.

Cantina Managed: Fixed; it now uses `UnmarshalJSON` from cubesigner repo which use `api.strictdecoder`.