coinspect

You build, we defend.

Babylon

Source Code Audit

Phase 1

June 2024

Babylon Phase 1
Source Code Audit

Version: v240805 Prepared for: Babylon Jun 2024

Security Assessment

1. Executive Summary

In **May 2024**, Babylon engaged Coinspect to perform a source code review of the code used to stake bitcoins during the Phase 1 of the Babylon mainnet deployment. The objective of the project was to evaluate the security of the different libraries and applications developed by Babylon to allow staking during this phase.

✓ Solved	⚠ Caution Advised	✗ Resolution Pending
High 0	High 0	High 0
Medium 5	Medium 0	Medium 0
Low 1	Low 1	Low 0
No Risk 5	No Risk 0	No Risk 0
Total **11**	Total **1**	Total **0**

`BP1-001` notes how staking and withdrawal transactions might not ever be included on the Bitcoin chain due to fixed fees. `BP1-004` describes how an attacker can prevent a victim from requesting unbonding transactions. `BP1-006` shows a possible denial of service from a single covenant member. `BP1-007` demonstrates how an attacker is able to conceal that they have withdrawn their stake. `BP1-008` notes how it is possible for a user to have their funds locked by mistake while using the Babylon-provided tooling.

2. Summary of Findings

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

2.1 Findings where caution is advised

These issues have been addressed, but their risk has not been fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating these issues or increasing their probability.

Findings with a risk of `None` pose no threat, but document an implicit assumption which must be taken into account. Once acknowledged, these are considered solved.

Id	Title	Risk
BP1-012	Attacker can prevent all stake and then unbound at no cost	Low

2.2 Solved issues & recommendations

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

Id	Title	Risk
BP1-001	Staking and withdrawal transactions might not be processed	Medium
BP1-004	Attacker can prevent victim from requesting signatures for unbonding transaction	Medium
BP1-006	Single rogue committee member can crash signer service	Medium

BP1-007	Attacker can conceal stake withdrawals	Medium
BP1-008	Users can get their funds locked when using Babylon tooling	Medium
BP1-011	Users allowed stake with finality providers with no commission disclosed	Low
BP1-002	Local storage data is inaccurate	None
BP1-003	Staking transaction cost might be elevated due to dust inputs	None
BP1-005	Insecure default RabbitMQ credentials	None
BP1-009	Signet values hardcoded	None
BP1-010	Users of Tomo wallet cannot double check transaction data	None

3. Scope

The review began on **May 15, 2024** and comprised 10 repositories. The review took place while the Babylon team was still working on the project, which led to a review process which consisted of a main review of the initial code plus subsequent `diffs` for each update provided.

The scope was always separated into two priorities set by the Babylon team. At the start of the review, the **top priority** repositories were:

1. https://github.com/babylonchain/babylon, `btc-staking` directory, at commit `7778c798e236a61c1dfe7d465e4d0cbded6483bb`
2. https://github.com/babylonchain/btc-staking-ts at commit `4510f3a683c20bf3b98f04e5c035d53224f29e47`
3. https://github.com/babylonchain/simple-staking at commit `7afa81c9092406a01f55249cd9b52625c0f4a4a6`
4. https://github.com/babylonchain/btc-staker/ `/cmd/stakercli/transactions.go` file, at commit `515865b3e5177e9cb29aec7d1ea3b1350d68dba8`
5. https://github.com/babylonchain/cli-tools/ at commit `bf17e4a3923b435ff885c88656526c8651227da3`
6. https://github.com/babylonchain/covenant-signer/, `docs` and `signerapp` directories, at commit `5b2b1e2f9cbe6b5c2ac575b4da27855a4272266f`

While the **second priority** repositories were:

1. https://github.com/babylonchain/staking-api-service/, at commit `3ce47dffc30784989f2bf3af59f9a8152cd19d8f`
2. https://github.com/babylonchain/staking-indexer/ at commit `9b7521d88fe73439d1b305c738319b95e39b530e`
3. https://github.com/babylonchain/staking-expiry-checker at commit `894bb046963ae6bea0008ff3682fd989914fe5ce`
4. https://github.com/babylonchain/staking-queue-client at commit `8cdd06780da3b0f9e9741a5f2928d5e41496e6da`

Note that when no specific files or directories were specified, the whole repository was in scope.

On **May 18**, the Babylon team updated the scope for two repositories and Coinspect continued the review these new commits:

1. https://github.com/babylonchain/btc-staking-ts/, new commit `b7ada8bab0fc371fa8c48c6807f64f99376f4178`
2. https://github.com/babylonchain/simple-staking/, new commit `17d0735442170e03d47e168b42b9d0fdf3b2417c`

On **May 29**, Babylon provided new commits for most of the repositories. On **June 3**, Coinspect started reviewing these new commits:

1. https://github.com/babylonchain/simple-staking new commit `45c92b36842fbc3365e0a9e0f049e0fa825a83af`
2. https://github.com/babylonchain/btc-staking-ts/, new commit `2483c97f6156d507f74ef4dcc814c67c29d44460`
3. https://github.com/babylonchain/covenant-signer/, new commit `a06a6b6d41c651e1a61a30692c00beb1305e3a78`
4. https://github.com/babylonchain/babylon, `btc-staking` directory new commit, `add420f074751cf53edea5b7a55cca3d34291f5b`
5. https://github.com/babylonchain/cli-tools/, new commit `98ad1d66e91ca8e477090a44492e1c68532dfeb8`
6. https://github.com/babylonchain/staking-indexer/, new commit `c7a3fbce12732856d66629d3bdc65fcd53246b6d`
7. https://github.com/babylonchain/staking-api-service/, new commit `219662164c1aabb64782582817bb9d782b7ec793`
8. https://github.com/babylonchain/staking-queue-client at commit `38c87828544a09d4beee74992b2a936b11d944b5`

The `covenant-signer` scope was also changed from only the `docs` and `signerapp` directories to comprise all the files in the repository.

On **June 14**, a second update was made to the repositories. These updates were pushed to private repositories which mirrored the public ones that were in use during the review until this point. For simplicity, the URLs refer to the public version of the repositories.

1. https://github.com/babylonchain/simple-staking new commit `9210576243d8aacd670a2e5d7689ccad7f74e8e8`
2. https://github.com/babylonchain/btc-staking-ts/, new commit `c482ac616683a6856729248db7eb091845d78e31`
3. https://github.com/babylonchain/covenant-signer/, new commit `91e4744bbe0bb440344354e380959d8126d9b82b`
4. https://github.com/babylonchain/babylon, `btc-staking` directory, new commit `340e7ba4948a902b00cd41be60d285cde5b4a093`
5. https://github.com/babylonchain/cli-tools/, new commit `d3921efd97bed74dbe9a3b8b578ab320e3460a52`
6. https://github.com/babylonchain/staking-indexer/, new commit `e4601c466de63193be52363d5cef6a560f261ccd`
7. https://github.com/babylonchain/staking-api-service/, new commit `280a8711ef50443286f7336f3d45cea67d23749c`
8. https://github.com/babylonchain/staking-queue-client at commit `3f07eacc102a7ea9861689a4028c825d4a67e854`
9. https://github.com/babylonchain/staking-expiry-checker, no new commits

The `babylon` repository, although updated, had no changes on the `btc-staking` directory, the only one in scope for this review.

On **June 18**, a third update was made by the Babylon team to some repositories. The new commits were first shared over private repositories and later moved to the public Babylon repositories.

1. https://github.com/babylonchain/simple-staking new commit `9040c942d0b811e880d284a69d8abbca0572614f`
2. https://github.com/babylonchain/btc-staking-ts/, new commit `6494df2b9f2c7a80578356659b1d24302e69dda2`
3. https://github.com/babylonchain/covenant-signer/, no new commits
4. https://github.com/babylonchain/babylon, `btc-staking` directory, new commit `add420f074751cf53edea5b7a55cca3d34291f5b`
5. https://github.com/babylonchain/cli-tools/, no new commits
6. https://github.com/babylonchain/staking-indexer/, new commit `c13b4f0dd1a57f5f327e5fee613bd41e1b923062`
7. https://github.com/babylonchain/staking-api-service/, new commit `4e6033a0860df23400611bad24ec72934545f374`
8. https://github.com/babylonchain/staking-queue-client no new commit
9. https://github.com/babylonchain/staking-expiry-checker, new commit `c04e2af4b38e363554b4a4b28485d484b837dbe3`

The `babylon` repository was again updated but no changes were made to the `btc-staking` directory, the only one in scope.

On **July 3**, Coinspect reviewed the fixes for each of the issues. All major issues have been fixed. `BP1-012` has been acknowledged and Babylon has stated they will take measures to mitigate the risk as much as possible. Each issue has now an updated status, showing exactly the commit where it was addressed.

4 Assessment

Babylon Phase 1 deployment aims to allow stakers to lock their bitcoins with the Babylon staking script and keep track of these stakes and their status. Any staker should also be able to unbond their stake, both on-request and after the stake has expired.

The reviewed system comprises several different projects. The system allows stakers to use the `simple-staking` frontend to create their staking transactions. Transactions are sent to the Bitcoin chain where they are picked up by the `indexer`. The `indexer` shares information about the transactions it finds with the `staking-api-service`, which reflects the status changes on the different transactions on its database. Transactions to unbond a delegation go to the `staking-api-service` *before* they are sent to the Bitcoin chain as they require signatures from the covenant committee besides the staker's signature. The `staking-api-service` stores them on its database, where they are picked up by a script that runs the `unbonding pipeline` periodically, processes each unbonding request and sends them to different `covenant-signer` servers ran by each covenant member. When this process is complete, the `unbonding pipeline` broadcasts the information to Bitcoin. The `unbonding pipeline` is implemented in the `cli-tools` repository.

Other repositories are utilities and dependencies for the ones mentioned in the paragraph above. For example, `staking-queue-client` handles connection and requests to and from RabbitMQ queues which `staking-api-service` and the `indexer` use to share information.

Due to the system's nature, there are two ways in which the stakers could potentially lose funds:

1. The lock-scripts are faulty, and somehow allow an attacker to steal the staker's coins or lock the bitcoins forever with no chance of recovering them.
2. The frontend or its API are compromised by hijacking the deploy process, dependencies, hosting or delivery service.

For the first scenario, Coinspect has verified that the scripts used in the Babylon mainchain are correct and that the new script builders match those created for the mainchain. The Babylon team has also created differential tests for the different implementations. No issues have been uncovered related to the scripts.

The second scenario was not directly covered by this review, as it is an operational risk and cannot be assessed from the codebase. For security-conscious users, the best way to generate the staking transaction is to use both the `btc-staker` and the frontend to generate two sets of independent transactions and make sure that the data in the transactions matches. This way, not a single component is fully trusted and there are less concerns about the authenticity of the frontend. Unfortunately, one of the supported wallets, `Tomo`, does not show any details to the user about the transaction-to-sign. This means `Tomo` users are more likely to fall for phishing attempts, as they cannot verify the transaction details. This risk is detailed in `BP1-010`.

Another threat is that the logic that generates transactions is faulty and allows stakers to generate staking transactions that are not valid, but have a correct unbond script and expiration logic. In contrast with the first scenario, this one would allow stakers to recover their locked coins eventually, at worst after their stake expires. During the review, one issue was discovered related to this scenario: `BP1-008`.

A related but different threat is that stakers are not able to `unbond` their transaction, not due to any fault in the transactions themselves, but due to covenant emulation committee members not signing their transactions. Even assuming the majority of the covenant emulation committee wants to cooperate, Coinspect found two issues (`BP1-004` and `BP1-006`) that would make them unable to do so. It is important to note that a majority of cooperating covenant emulation committee members is an assumption for this review (see `4.1 Security Assumptions`). In practice, a majority of covenant emulation committee members could become rogue. If this were to happen, stakers could recover their locked coins after their stake expires.

Other threats to the system would have a recoverable consequence. For example, if the system has a bug by which the indexer does not parse staking transactions this would cause only temporary impact and no loss of funds: the authoritative source of truth would still be the Bitcoin chain, where data will be present in its raw form and available to be ingested and processed after the problems are fixed. `BP1-007` describes one issue of this category.

Lastly, the system was under active development in parallel with the review process, leading to some components not being fully functional. `BP1-009`, for example, describes a bug by which different parts of the system had hardcoded values which made it unusable on mainnet. The Babylon team was still developing the system during this review, which explains these issues and the several updates made to the scope of the review. Coinspect was able to continue the review as the bugs were fixed and examine the additional changes.

It is important to note that users need to trust the party that is serving the frontend and running the backend. Users are free to trust a provider of their choosing, as anyone can host the reviewed applications as long as they are willing to set up the infrastructure. Information on `commission` provided by the finality providers must also be trusted, as there is no Babylon mainchain yet where to check these commission rates, and finality providers are able to set them to an arbitrary number when the Babylon blockchain is deployed. In the worst case scenario, a finality provider can change their `comission` to the highest one allowed by the Babylon blockchain. Users would regain access to their coins after `unbonding_time` and would be able to stake against with an honest finality provider. Users should monitor the status of the finality provider in the Babylon blockchain when it deploys to confirm the `commission` they are being charged.

Another consideration to take into account is that it is entirely possible for finality providers to simply not provide any finality services to the Babylon blockchain when it is deployed. Note that this means that the TVL for Phase 1 may or may not become TVL for Phase 2, as additional steps are required from the finality provider for Phase 1's Bitcoin stakes to become accepted for Phase 2.

It is worth mentioning that the exact parameters to be used were not defined at the time of the review. Coinspect assumed that important security parameters such as `confirmation_depth` are to be set to reasonable values so as not to allow trivial attacks (i.e: a `confirmation_depth` of one would allow reorganization-related attacks, having a single `covenant_member` would defeat the purpose of a committee, etc.). Users are encouraged to check the `params` chosen before using the system.

4.1 Security assumptions

Coinspect made the following assumptions during the review:

1. The Bitcoin network is safe and live.
2. A majority of covenant emulation committee members are online and respond to signing requests in a timely manner.
3. The provider that hosts the frontend and API components of the web applications is trusted by the user.
4. The wallet providers correctly protect the user's signature and private key and don't modify the user's transaction.
5. The provider that hosts the components connects the indexer to a Bitcoin node that reports the actual mainchain data.
6. The `confirmation_depth` parameter is bigger or equal than six and the covenant emulation committee has more than a single member

5. Detailed Findings

BP1-001

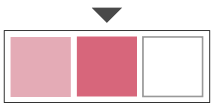
Staking and withdrawal transactions might not be processed

Status
Solved

✓

Resolution
Fixed

Risk
Medium



Impact
Low

Likelihood
High

Location

```
simple-staking:/src/app/page.tsx
```

Description

Staking and withdrawal transactions might not get included in the Bitcoin ledger due to hardcoded transaction fees.

The snippet below was extracted from the staking DApp, which predefines the staking and withdrawal transaction's fees.

```
const stakingFee = 500;
const withdrawalFee = 500;
```

The Bitcoin network fees are dynamic, fluctuating based on the current network load. Consequently, during periods of high network congestion, transactions with predefined fees may not be processed.

Recommendation

Use dynamic fees instead, to adjust transaction fees based on real-time network usage.

Status

Fixed in commits `2483c97f6156d507f74ef4dcc814c67c29d44460` and `f6da09b8cf830e0f0dba5c372c12e49057900da4`. Fees are now calculated with the user's wallet.

BP1-002

Local storage data is inaccurate

Status
Solved

✓

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
-

Location

`simple-staking:src/utils/local_storage/toLocalStorageDelegation.ts`

Description

The delegation stored in local storage will contain hardcoded data that is not accurate. For example, the `staking_tx.start_height` will be hardcoded to zero. Currently this has no concrete impact on the application, as `delegation` data is requested from the backend again before creating `unbond` or `withdrawal` transactions and the zero data is never used.

```
export const toLocalStorageDelegation = (
  stakingTxHashHex: string,
  stakerPkHex: string,
  finalityProviderPkHex: string,
  stakingValue: number,
  stakingTxHex: string,
  timelock: number,
): Delegation => ({
  staking_tx_hash_hex: stakingTxHashHex,
  staker_pk_hex: stakerPkHex,
  finality_provider_pk_hex: finalityProviderPkHex,
  state: DelegationState.PENDING,
  staking_value: stakingValue,
  staking_tx: {
    tx_hex: stakingTxHex,
    output_index: 0,
    start_timestamp: new Date().toISOString(),
    start_height: 0,
    timelock,
  },
  is_overflow: false,
});
```

Recommendation

Record the exact height where the staking transaction was submitted in the `Delegation` object.

Status

Fixed in commit `81cc1a128bac97f071af70d3d24cee9fefa17e8b` on `simple-staking`. Delegations are now fetched from the API more frequently. Although hardcoded values remain, this does not impact `unbond` nor `withdrawal` transaction building.

BP1-003

Staking transaction cost might be elevated due to dust inputs

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
-

Location

```
simple-staking:src/app/page.tsx:126
simple-staking:src/utils/wallet/okx_wallet.ts:139
simple-staking:src/utils/mempool_api.ts:124
```

Description

The staking transaction might contain dust inputs, causing the staking transaction fee to cost more than usual.

A dust input value refers to a very small amount of bitcoin that is considered uneconomical to spend or process. This is because the transaction fee required to move or spend such a small amount is higher than the value of the dust itself.

The in-scope `simple-staking` repository code did not show any reference to dust checking.

Note that since outputs are selected from larger to smaller values, this scenario is less likely to occur.

Recommendation

Check for dust inputs and exclude them.

Status

Fixed. The likelihood of using dust inputs is small. On the other hand, a check has been added on commit `cde2f25f581dd8e591f39e6f9c7224950c578fc2` which avoids creating dust outputs.

BP1-004

Attacker can prevent victim from requesting signatures for unbonding transaction

Status
Solved

✓

Resolution
Fixed

Risk
Medium

Impact
Medium

Likelihood
Medium

Location

```
staking-api-service:internal/services/unbonding.go
```

Description

An attacker can prevent a user from requesting the signatures needed for their unbond transaction from the covenant emulation committee members.

Consider the flow a user needs to go through to `unbond`: they need to provide the `staking-api-service` with their signed unbond transaction so that the service can request a signature from the covenant emulation committee members. The `staking-api-service` will write the `stakingTxHash`, the `unbondTxHash` and the `unbondTxHex` to the database.

This data will be picked up by the `cli-tools` unbond-pipeline functionality, which will check the data, sign the transaction and send it to Bitcoin.

Nevertheless, at no moment in the process will the system check that `unbondTxHash` actually matches the `unbondTxHex`. This, combined with the fact that the `unbondTxHash` is used as a primary identifier for the database means that an attacker can send a victim's `unbondTxHash` with an `unbondTxHex` of theirs, which will stop an honest user from requesting an unbond of their actual `(unbondTxHash, unbondTxHex)` pair.

If an attacker leverages this, the victim will get an `unbonding transaction already exists` error when trying to unbond.

Recommendation

Make sure that the `unbondTxHash` matches the `unbondTxHex` by performing a double-SHA256 of the `unbondTxHex`.

Proof of Concept

This proof of concept is intended to be added to `unbonding_test.go`. For simplicity and because it is not relevant for the PoC, the `StakerSignedSignatureHex` is not a valid signature: the validations below `// 4. verify the signature` in `VerifyUnbondingRequest` can be commented out. Note that the commented-out validations use only the staker unbond transaction, not its hash, so they are not relevant for the issue described above.

```
func TestAttackerShouldNotBeAbleToPreventUnbondRequest(t *testing.T) {
   // Let us create two staking events, one by the victim and one by
the
   // attacker. Both need to have staked
   victimStakingEvent := &client.ActiveStakingEvent{
       EventType:             client.ActiveStakingEventType,
       StakingTxHashHex:
"379155a9a081771ca64b5f73d3bf9d7611eb767d5a9f5c40aa6d769576fd35bc",
       StakerPkHex:
"02398a9d826ff189bcbf0f46291fb1efd78012b64741cc019f9046a123876b4e",
       FinalityProviderPkHex:
"a04d7107e796b3e0cc359c5683a3651de852cde5f9a7973292483ab0a89a3e2c",
       StakingValue:          110000,
       StakingStartHeight:    200,
       StakingStartTimestamp: time.Now().Unix(),
       StakingTimeLock:       100,
       StakingOutputIndex:    1,
       StakingTxHex:          "abcdef1234567890",
       IsOverflow:            false,
   }
   attackerStakingEvent := &client.ActiveStakingEvent{
       EventType:             client.ActiveStakingEventType,
       StakingTxHashHex:
"379155a9a081771ca64b5f73d3bf9d7611eb767d5a9f5c40aa6d769576fd35bd",
       StakerPkHex:
```

```
"02398a9d826ff189bcbf0f46291fb1efd78012b64741cc019f9046a123876b4e",
       FinalityProviderPkHex:
"184186d905450d9f0039ff91a15fe269db12d6c8eb5582cdd626a0c3f1379e11",
       StakingValue:          110000,
       StakingStartHeight:    200,
       StakingStartTimestamp: time.Now().Unix(),
       StakingTimeLock:       100,
       StakingOutputIndex:    1,
       StakingTxHex:          "abcdef1234567890babe",
       IsOverflow:            false,
   }


   // The attacker's unbond request looks exactly like a normal unbond
request,
   // except they put the victim's UnbondingTxHashHex. The attacker
   // can predict the victim's UnbondingTxHashHex because the format of
an unbonding
   // transaction is quite restricted and the address used by victim
staker is known (public in the blockchain)
   attackerUnbondRequest := handlers.UnbondDelegationRequestPayload{
       StakingTxHashHex:   attackerStakingEvent.StakingTxHashHex,
       UnbondingTxHashHex:
"47ed9d80620b118c4ca558c9dd51b59fb03598eeb1674fbe57c6f7dfbbd97c7e",
       UnbondingTxHex:
"0200000000101bd35fd7695766daa405c9f5a7d76eb11769dbfd3735f4ba61c7781a0
a955913701000000ffffffff01905f0100000000022" +
           // Output script

"5120f6217d6f1c6077ec82304a9acd30d7144f0f0a63873696a120c6f92688f2ce25"
+
           // Rest

"0440d780fa0e6dd463db09ad35df89c2be1eb23ad1a3d6b4dd7307894941621189a5f6
94d3d3eb059d7af6918695dd725c1de5abe80fb6d96f613e213af4aba303b340d0eaabc
52fb941616e2da0c4a506e2694a7f82f73a4c29fef031e6a6de7982431a5eabf40ed456
74c8f72d6f0c93c68dcca6706ff6009e26914f68cfa1d31411ce20b0f61bfae41af83d8
51a8211f82df861e93b3d39fd40a9b0e7f83bb655dad70bad2057349e985e742d5131e1
e2b227b5170f6350ac2e2feb72254fcc25b3cee21a18ac2059d3532148a597a2d05c039
5bf5f7176044b1cd312f37701a9b4d0aad70bc5a4ba20a5c60c2188e833d39d0fa798ab
3f69aa12ed3dd2f3bad659effa252782de3c31ba20c8ccb03c379e452f10c81232b41a1
ca8b63d0baf8387e57d302c987e5abb8527ba20ffeaec52a9b407b355ef6967a7ffc15f
d6c3fe07de2844d61550475e7a5233e5ba539c61c050929b74c1a04954b78b4b6035e97
a5e078a5a0f28ec96d547bfee9ace803ac07f9cd831c337b41bc9c6768468ea1eadb6f4
3285948e24ccc789629a72ddc116b83fc844969081daa83469d937e5aad55b7ae93c2df
837496391c5e8d927a06700000000",
       StakerSignedSignatureHex:
"d0eaabc52fb941616e2da0c4a506e2694a7f82f73a4c29fef031e6a6de7982431a5eab
f40ed45674c8f72d6f0c93c68dcca6706ff6009e26914f68cfa1d31411",
   }


   // Setup: setup a test server where both victim and attacker have
staked
   events := []client.ActiveStakingEvent{*victimStakingEvent,
*attackerStakingEvent}
   testServer := setupTestServer(t, nil)
   defer testServer.Close()
   err := sendTestMessage(testServer.Queues.ActiveStakingQueueClient,
       []*client.ActiveStakingEvent{victimStakingEvent})
```

```
    require.NoError(t, err)
    err = sendTestMessage(testServer.Queues.ActiveStakingQueueClient,
        []*client.ActiveStakingEvent{attackerStakingEvent})
    require.NoError(t, err)
    time.Sleep(2 * time.Second)


    // Make sure that both victim and staker are elligible to unbond
    for _, e := range events {
        eligibilityUrl := testServer.Server.URL +
unbondingEligibilityPath +
            "?staking_tx_hash_hex=" +
            e.StakingTxHashHex
        resp, err := http.Get(eligibilityUrl)
        assert.NoError(t, err, "making GET request to unbonding
eligibility check endpoint should not fail")
        defer resp.Body.Close()


        assert.Equal(t, http.StatusOK, resp.StatusCode, "expected HTTP
200 OK status")
    }


    // At this point we have two valid stakings and
    // both are elligible for unbonding
    // The attacker will now request an unbound request with _their_
unbond transaction
    // but the victim's unbond transaction hash. We will later show this
makes it impossible
    // for the victim to request _their_ unbond transaction
    unbondingUrl := testServer.Server.URL + unbondingPath
    requestBodyBytes, err := json.Marshal(attackerUnbondRequest)
    assert.NoError(t, err, "marshalling request body should not fail")
    resp, err := http.Post(unbondingUrl, "application/json",
bytes.NewReader(requestBodyBytes))
    assert.NoError(t, err, "making POST request to unbonding endpoint
should not fail")
    defer resp.Body.Close()
    assert.Equal(t, "202 Accepted", resp.Status, "attacker request
rejected")


    victimUnbondRequest := handlers.UnbondDelegationRequestPayload{
        StakingTxHashHex:         victimStakingEvent.StakingTxHashHex,
        UnbondingTxHashHex:
"47ed9d80620b118c4ca558c9dd51b59fb03598eeb1674fbe57c6f7dfbbd97c7e",
        UnbondingTxHex:
"02000000000101bc35fd7695766daa405c9f5a7d76eb11769dbfd3735f4ba61c7781a0
a95591370100000000ffffffff01905f0100000000000022" +
"5120a7a317ad1f22f47004590ce93e9dc7176760964c17c28c0266e367426e31aae6"
+
"0440d780fa0e6dd463db09ad35df89c2be1eb23ad1a3d6b4dd7307894941621189a5f6
94d3d3eb059d7af6918695dd725c1de5abe80fb6d96f613e213af4aba303b340d0eaabc
52fb941616e2da0c4a
506e2694a7f82f73a4c29fef031e6a6de7982431a5eabf40ed45674c8f72d6f0c93c68d
cca6706ff6009e26914f68cfa1d31411ce20b0f61bfae41af83d851a8211f82df861e93
b3d39fd40a9b0e7f83bb655dad70bad2057349e985e742d5131e1e2b227b5170f6350ac
2e2feb72254fcc25b3cee21a18ac2059d3532148a597a2d05c0395bf5f7176044b1cd31
2f37701a9b4d0aad70bc5a4ba20a5c60c2188e833d39d0fa798ab3f69aa12ed3dd2f3ba
d659effa252782de3c31ba20c8ccb03c379e452f10c81232b41a1ca8b63d0baf8387e57
d302c987e5abb8527ba20ffeaec52a9b407b355ef6967a7ffc15fd6c3fe07de2844d615
50475e7a5233e5ba539c61c050929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d
547bfee9ace803ac07f9cd831c337b41bc9c6768468ea1eadb6f43285948e24ccc78962
9a72ddc116b83fc844969081daa83469d937e5aad55b7ae93c2df837496391c5e8d927a
06700000000",
       StakerSignedSignatureHex:
"d0eaabc52fb941616e2da0c4a506e2694a7f82f73a4c29fef031e6a6de7982431a5eab
f40ed45674c8f72d6f0c93c68dcca6706ff6009e26914f68cfa1d31411",
   }
   requestBodyBytes, err = json.Marshal(victimUnbondRequest)
   assert.NoError(t, err, "marshalling request body should not fail")
   resp, err = http.Post(unbondingUrl, "application/json",
bytes.NewReader(requestBodyBytes))
   assert.NoError(t, err, "making POST request to unbonding endpoint
should not fail")
   defer resp.Body.Close()
   assert.Equal(t, "202 Accepted", resp.Status, "victim request
rejected")
}
```

Status

Fixed in commit `8feea7db74a5c938acf31d0c72e1b7fec759fd1d`. Hashes are now compared to make sure the unbonding data and the unbonding hash provided match.

BP1-005

Insecure default RabbitMQ credentials

Status
Solved

✓

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
–

Location

`staking-queue-client:config/queue.go`

Description

The `staking-queue-client`, an interface to interact with RabbitMQ queues, has a `DefaultQueueConfig` with insecure values for user and password:

```
    defaultQueueUser                    = "user"
    defaultQueuePassword                = "password"
```

This issue is only informational because the `staking-api-service`, which creates the queue client, does not use the default values. It nevertheless represents a risk for any other projects that might use this configuration provided by the library.

Recommendation

Remove the `DefaultQueueConfig` or make the `Validate()` method check for the default user and password and reject configurations that leave these fields as the default.

Status

Acknowledged. These are set only as dummy-defaults. Babylon has stated they will use other credentials in production.

BP1-006

Single rogue committee member can crash signer service

Status
Solved

✓

Resolution
Fixed

Risk
Medium

🟥🟥⬜

Impact
High

Likelihood
Low

Location

`covenant-signer:signerservice/client.go`

Description

A single rogue covenant emulation committee member can crash the covenant signer process by sending a big response to the `sign-unbonding-tx` request.

The root cause of the issue is the use of `io.ReadAll` for the response body. `io.ReadAll` is a limit-less reader which will try to put all the data from the body in a buffer. A rogue covenant emulation committee member can send an arbitrary amount of data in the request, triggering an out-of-memory error.

The attacker would be restricted only by the request timeout, which by default is only 2 seconds. Nevertheless, several EC2 instances have a network bandwidth of several Gb per second. For example, by using a `m7g.16xlarge` instance, an attacker could send up to 30Gb per second to the covenant-signer; about 60Gb total. This would be enough data to cause an out-of-memory error in most configurations.

Recommendation

Do not use `ReadAll` to read data from the body. Instead, use a io.LimitReader with a sensible limit to the amount of bytes it can read.

Status

Fixed in commit `b8609d381780daa00ab38a5eb1bfb4150c75fcdf`. The reader is now implemented with `http.MaxBytesReader`, limiting the size of the incoming data.

BP1-007

Attacker can conceal stake withdrawals

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
Low

Likelihood
High

Location

`staking-indexer/indexer.go`

Description

An attacker can prevent a staked transaction from being recorded as `withdrawn` by staking twice, unbonding both transactions, and creating a single withdrawal transaction that uses the two unbonding transactions as inputs. This results in one of the staking transactions not being recorded as `withdrawn`.

To exploit this issue, one needs to consider that when processing new blocks, the indexer searches for spent staked inputs on each transaction:

```
// handleConfirmedBlock iterates all the tx set in the block and
// parse staking tx, unbonding tx, and withdrawal tx if there are any
func (si *StakingIndexer) handleConfirmedBlock(b *types.IndexedBlock)
error {
   for _, tx := range b.Txs {
       msgTx := tx.MsgTx()
```

```
        // 1. try to parse staking tx
    ...


        // 2. not a staking tx, check whether it spends a stored staking
tx
        stakingTx, spentInputIdx := si.getSpentStakingTx(msgTx)
```

The problem appears inside the `getSpentStakingTx`. The function returns a possible staking transaction spent if it exists:

```
// getSpentStakingTx checks if the given tx spends any of the stored
staking tx
// if so, it returns the found staking tx and the spent staking input
index,
// otherwise, it returns nil and -1
func (si *StakingIndexer) getSpentStakingTx(tx *wire.MsgTx)
(*indexerstore.StoredStakingTransaction, int) {
   for i, txIn := range tx.TxIn {
        maybeStakingTxHash := txIn.PreviousOutPoint.Hash
        stakingTx, err := si.GetStakingTxByHash(&maybeStakingTxHash)
        if err != nil {
            continue
        }


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


        return stakingTx, i
    }


    return nil, -1
}
```

This function returns the first staking transaction that is used as an input. If multiple staking transactions are used as inputs, they are ignored. This means that the staking transaction used as a second input would not be marked as withdrawn.

The impact of this vulnerability is heavily dependent on how the Bitcoin data is used when the Babylon blockchain launches. Depending on how the data is used, this could impact staking reward or TVL counts.

Recommendation

Check for all the spent inputs within the getSpentStakingTx and return an array instead.

Proof of Concept

The proof of concept shows how GetSpentUnbondingTx only returns the first unbond transaction. It is intended to be added to indexer/indexer_test.go and testutils/datagen/staking.go.

The new test requires getSpentUnbondingTx to be a public method (i.e.: changing it to GetSpentUnbondingTx)

```
// staking.go function


func GenerateWithdrawalTxFromUnbondingTwoInputs(t *testing.T, r
*rand.Rand, params *types.GlobalParams, stakingData *TestStakingData,
unbondingTxHash *chainhash.Hash, anotherUnbondingTxHash
*chainhash.Hash) *btcutil.Tx {
   // build and send withdraw tx from the unbonding tx
   unbondingInfo, err := btcstaking.BuildUnbondingInfo(
       stakingData.StakerKey,
       []*btcec.PublicKey{stakingData.FinalityProviderKey},
       params.CovenantPks,
       params.CovenantQuorum,
       params.UnbondingTime,
       stakingData.StakingAmount.MulF64(0.9),
       &chaincfg.SigNetParams,
   )
   require.NoError(t, err)
   withdrawSpendInfo, err := unbondingInfo.TimeLockPathSpendInfo()
   require.NoError(t, err)


   withdrawalTx := wire.NewMsgTx(2)
   witness, err := btcstaking.CreateWitness(withdrawSpendInfo, [][]byte{})
   require.NoError(t, err)


   withdrawalTx.AddTxIn(wire.NewTxIn(wire.NewOutPoint(unbondingTxHash,
0), nil, witness))

withdrawalTx.AddTxIn(wire.NewTxIn(wire.NewOutPoint(anotherUnbondingTxHash, 0), nil, witness))
   // add a dump input
   randomOutput := &wire.OutPoint{
       Hash:  chainhash.HashH(bbndatagen.GenRandomByteArray(r, 10)),
       Index: r.Uint32(),
   }
   withdrawalTx.AddTxIn(wire.NewTxIn(randomOutput,
bbndatagen.GenRandomByteArray(r, 10), nil))
```

```
   return btcutil.NewTx(withdrawalTx)
}


// indexer_test.go test
func TestDoubleInputWithdrawalShouldBeRegistered(t *testing.T) {
   r := rand.New(rand.NewSource(0x42))


   homePath := filepath.Join(t.TempDir(), "indexer")
   cfg := config.DefaultConfigWithHome(homePath)


   sysParamsVersions := datagen.GenerateGlobalParamsVersions(r, t)


   db, err := cfg.DatabaseConfig.GetDbBackend()
   require.NoError(t, err)
   chainUpdateInfoChan := make(chan *btcscanner.ChainUpdateInfo)
   mockBtcScanner := NewMockedBtcScanner(t, chainUpdateInfoChan)
   stakingIndexer, err := indexer.NewStakingIndexer(cfg, zap.NewNop(),
NewMockedConsumer(t), db, sysParamsVersions, mockBtcScanner)
   require.NoError(t, err)
   defer func() {
       err = db.Close()
       require.NoError(t, err)
   }()


   // Select the first params versions to play with
   params := sysParamsVersions.ParamsVersions[0]
   // 1. generate and add a valid staking tx to the indexer
   oneStakingData := datagen.GenerateTestStakingData(t, r, params)
   _, oneStakingTx := datagen.GenerateStakingTxFromTestData(t, r,
params, oneStakingData)


   anotherStakingData := datagen.GenerateTestStakingData(t, r, params)
   _, anotherStakingTx := datagen.GenerateStakingTxFromTestData(t, r,
params, anotherStakingData)


   // For a valid tx, its btc height is always larger than the
activation height
   mockedHeight := uint64(params.ActivationHeight) + 1


   // Process oneStakingData/tx
   err = stakingIndexer.ProcessStakingTx(
       oneStakingTx.MsgTx(),
       getParsedStakingData(oneStakingData, oneStakingTx.MsgTx(),
params),
       mockedHeight, time.Now(), params)


   require.NoError(t, err)
   oneStoredStakingTx, err :=
stakingIndexer.GetStakingTxByHash(oneStakingTx.Hash())
   require.NoError(t, err)
   require.NotNil(t, oneStoredStakingTx)
   // Process anotherStakingData/tx
   err = stakingIndexer.ProcessStakingTx(
       anotherStakingTx.MsgTx(),
       getParsedStakingData(anotherStakingData,
anotherStakingTx.MsgTx(), params),
       mockedHeight, time.Now(), params)


   require.NoError(t, err)
   anotherStoredStakingTx, err :=
stakingIndexer.GetStakingTxByHash(anotherStakingTx.Hash())
   require.NoError(t, err)
   require.NotNil(t, anotherStoredStakingTx)


   // 2. generate unbonding txs for both
   oneUnbondingTx := datagen.GenerateUnbondingTxFromStaking(t, params,
oneStakingData, oneStakingTx.Hash(), 0)
   isValid, err :=
stakingIndexer.IsValidUnbondingTx(oneUnbondingTx.MsgTx(),
oneStoredStakingTx, params)
   require.NoError(t, err)
   require.True(t, isValid)


   anotherUnbondingTx := datagen.GenerateUnbondingTxFromStaking(t,
params, anotherStakingData,
       anotherStakingTx.Hash(), 0)
   isValid, err =
stakingIndexer.IsValidUnbondingTx(anotherUnbondingTx.MsgTx(),
anotherStoredStakingTx, params)
   require.NoError(t, err)
   require.True(t, isValid)


   err = stakingIndexer.ProcessUnbondingTx(oneUnbondingTx.MsgTx(),
       oneStakingTx.Hash(), mockedHeight+10, time.Now(), params)
   require.NoError(t, err)


   err = stakingIndexer.ProcessUnbondingTx(anotherUnbondingTx.MsgTx(),
       anotherStakingTx.Hash(), mockedHeight+10, time.Now(), params)
   require.NoError(t, err)


   withdrawTxFromUnbonding :=
datagen.GenerateWithdrawalTxFromUnbondingTwoInputs(t, r, params,
oneStakingData, oneUnbondingTx.Hash(), anotherUnbondingTx.Hash())
   err =
stakingIndexer.ValidateWithdrawalTxFromUnbonding(withdrawTxFromUnbondin
g.MsgTx(), oneStoredStakingTx, 0, params)
   require.NoError(t, err)


   recoveredUnbondingTx, spendingInputIdx :=
stakingIndexer.GetSpentUnbondingTx(withdrawTxFromUnbonding.MsgTx())
   require.True(t, spendingInputIdx == 0)
   require.True(t, recoveredUnbondingTx.Tx.TxHash() ==
*oneUnbondingTx.Hash())
}
```

Status

Fixed in commit `d9e462c38d9784d169dc40c0685803f134f5b8ac`. The methods that look for spent transactions do not return on the first-input-matched and instead loop over all inputs used looking for matches. They then return a slice containing all matches.

BP1-008

Users can get their funds locked when using Babylon tooling

Status
Solved

✓

Resolution
Fixed

Risk
Medium

▨ ■ □

Impact
High

Likelihood
Low

Location

```
cli-tools:cmd/createStakingTxCmd.go
btc-staker:cmd/stakercli/transaction/transactions.go
```

Description

Users can get their funds locked for up to `timelock` time when using the `cli-tools` or the `btc-staker` projects. This is because both repositories fail to validate the minimum and maximum stake values, allowing users to attempt staking values out of the valid bounds.

This can be seen in the methods `createStakingTxCmd` of `cli-tools` and `createPhase1UnbondingTransaction` of `btc-staker`.

Such stakes would produce transactions that lock funds, but that do not count as valid stake; making it a net loss for the user. The funds will be recoverable only after `timelock` passes.

Covenants will not sign unbonding transactions for those stakes either, preventing the unbonding path from executing.

Recommendation

Request the current staking parameters from a Babylon node or backend and prevent users from building invalid staking transactions.

Proof of concept

The issue can be replicated by using the babylon-btcstaking-phase1-demo repository. The lines starting from line 293 should read:

```
move_to_block 115
## At height 115, we send 1 staking tx with a very low value in BTC
## and one with a valid value
create_staking_tx 100000000 1000 0 # 1 BTC
create_staking_tx 1000 1000 1 # 1 BTC


move_next_block # 116
move_next_block # 116
## At height 117, only the valid value staking tx is considered,
## even though the other one was sent to the network too
check_mongoDB_info $(cat $DIR/100000000/tx_id) "active" "false"
check_mongoDB_info $(cat $DIR/1000/tx_id) "active" "false"
```

The script will never finish: the transaction with value `1000` will never be found in the Mongo database.

Status

Fixed in commit `c853313ad8bc0c6058ad3272662e88f46b47c0e6` for `btc-staker`, which now has commands to create transactions following the parameters specified in a global params file.

`cli-tools` has been left as is, as the commands to create transactions are intended for testing scenarios and has documentations that explicitly states so.

BP1-009

Signet values hardcoded

Status

Solved

Resolution

Fixed

Risk

None

Impact

Recommendation

Likelihood

-

Location

`simple-staking`

Description

Several pieces of the codebase have hardcoded references to `signet` being the network to use. If not updated before the public release, these references can make users send transactions on the wrong network.

For example, in `simple-staking:src/utils/wallet/providers/okx_wallet.ts`, the network is hardcoded as `signet`:

```
getNetwork = async (): Promise<Network> => {
  return Network.SIGNET;
};
```

Similarly for the `bitget_wallet` at `simple-staking:src/utils/wallet/providers/bitget_wallet.ts`

```
await this.provider?.switchNetwork("signet");
```

Recommendation

Update the hardcoded network references before release.

Status

Fixed in commit `9040c942d0b811e880d284a69d8abbca0572614f`. The repository is now prepared to handle different networks, including mainnet.

BP1-010

Users of Tomo wallet cannot double check transaction data

Status
Solved

✓

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
-

Location

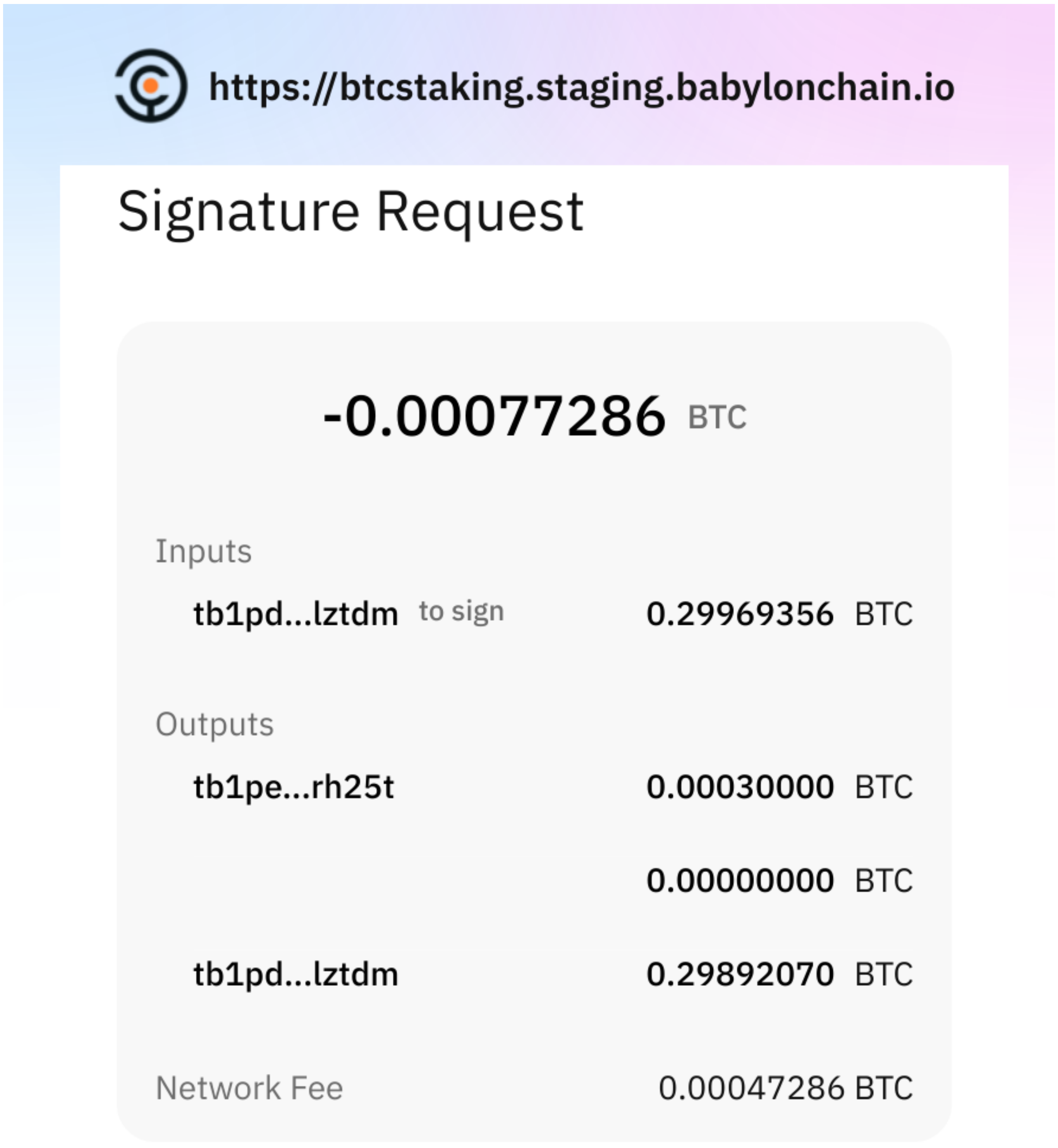
```
simple-staking
```

Description

Users of the Tomo wallet are forced to sign their transaction without seeing the script which locks their coins, as the wallet extension does not show any detail about the scripts.

This means users of the Tomo wallet are forced to trust the Babylon frontend with no possibilities to double-check the script with the ones provided by the alternative tools provided by Babylon or an independent script-generator. In case of a compromise of the frontend or a phishing attempt, users have no last-measure recourse.

To understand the issue, consider the sign transaction screen presented to the user by Tomo:



https://btcstaking.staging.babylonchain.io

Signature Request

-0.00077286 BTC

Inputs

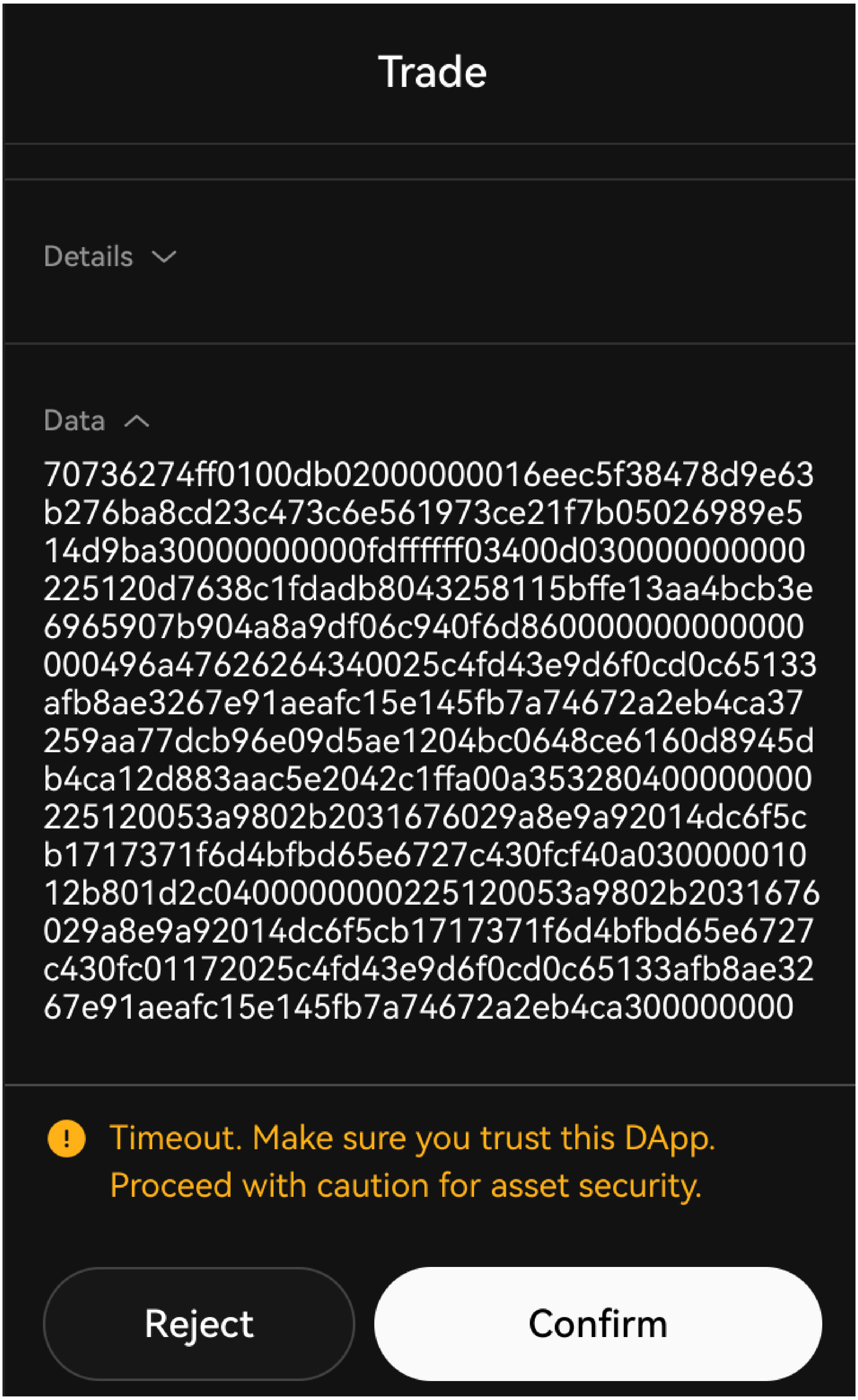
tb1pd...lztdm	to sign	0.29969356 BTC

Outputs

tb1pe...rh25t	0.00030000 BTC
	0.00000000 BTC
tb1pd...lztdm	0.29892070 BTC

Network Fee 0.00047286 BTC

Compare the screen to the data shown to other wallet providers which *do* show the data to be signed, such as OKX.



```
70736274ff0100db02000000016eec5f38478d9e63
b276ba8cd23c473c6e561973ce21f7b05026989e5
14d9ba3000000000fdffffff03400d030000000000
225120d7638c1fdadb8043258115bffe13aa4bcb3e
6965907b904a8a9df06c940f6d8600000000000000
000496a47626264340025c4fd43e9d6f0cd0c65133
afb8ae3267e91aeafc15e145fb7a74672a2eb4ca37
259aa77dcb96e09d5ae1204bc0648ce6160d8945d
b4ca12d883aac5e2042c1ffa00a353280400000000
225120053a9802b2031676029a8e9a92014dc6f5c
b1717371f6d4bfbd65e6727c430fcf40a030000010
12b801d2c0400000000225120053a9802b2031676
029a8e9a92014dc6f5cb1717371f6d4bfbd65e6727
c430fc01172025c4fd43e9d6f0cd0c65133afb8ae32
67e91aeafc15e145fb7a74672a2eb4ca300000000
```

Timeout. Make sure you trust this DApp. Proceed with caution for asset security.

Recommendation

Warn Tomo wallet users about the risks of not double checking the data being signed. Generally encourage users to double-check the script-to-be-signed with an independent implementation.

Status

Acknowledged. Babylon has communicated the issue to Tomo. Tomo users should be aware of this limitation.

BP1-011

Users allowed stake with finality providers with no commission disclosed

Status

Solved

Resolution

Fixed

Risk

Low

Impact

Low

Likelihood

Low

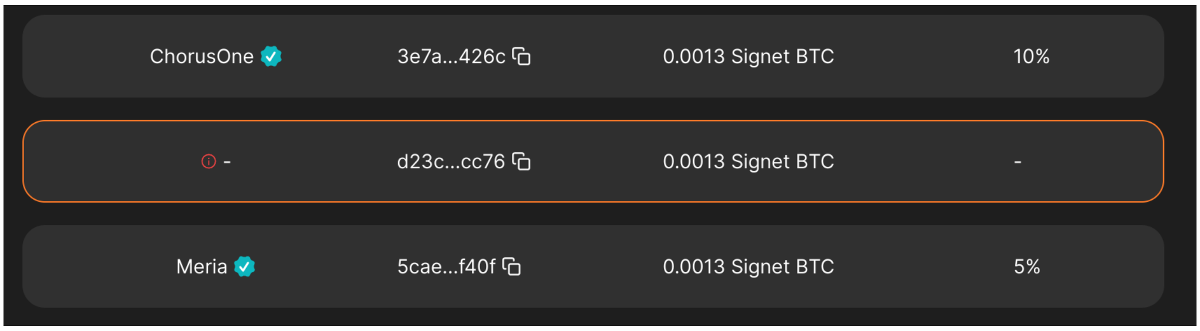
Location

`simple-staking`

Description

Finality providers with no commission nor name are displayed in the frontend. This makes it impossible for users to have the information needed to decide whether they prefer one finality provider over another.

What is more, due to the design chosen, there is ambiguity as the `-` symbol can be interpreted as "no" commission.



ChorusOne ✓	3e7a...426c	0.0013 Signet BTC	10%
ⓘ -	d23c...cc76	0.0013 Signet BTC	-
Meria ✓	5cae...f40f	0.0013 Signet BTC	5%

Recommendation

Do not allow finality providers to advertise with no name nor commission. Have a clear strategy for making sure that the commission displayed on the frontend is later respected in the mainchain.

Status

Fixed in commit `e417560038fc3d6ae51e18cc1b1afe0f4f68fb6b` on the `simple-staking` repository. Staking providers that have not provided a moniker and a commission are not selectable by the user.

BP1-012

Attacker can prevent all stake and then unbound at no cost

Status
Caution Advised

Resolution
Acknowledged

Risk
Low

Impact
Low

Likelihood
Low

Description

An attacker with enough capital can prevent others from staking by staking their own funds and forcing all other staking transactions to be considered `overflow`.

Due to the way the system is designed, the attacker can simply unbond after the unbonding time has passed, recovering their capital. Note that no economic security was provided to the Babylon blockchain, even if the attacker made it look like capital was being deployed for staking.

This means that the attacker manages to:

1. Perform a denial-of-service against honest stakers
2. Make the system calculate a misleading TVL count which will never be put to use by the Babylon blockchain

The attack only requires the attacker to have or be able to loan enough BTC to reach the `staking_cap`. This value is defined in the `Params` of Babylon. It is unknown what this value will be on the release version of the software at this time (see `Assessment` section).

The impact of this attack is lowered because the `staking_cap` can be dynamically set if such an attack is detected.

Recommendation

Deprecate the `staking_cap` parameter and support only the `staking_height` parameter.

Status

Acknowledged. As a mitigation, the Babylon team states that the parameters related to the staking cap and maximum stake amount per transaction will be set so that the attack is too expensive to perform. With a correct combination of the two, an attacker would need to spend a very high amount of bitcoins to carry out the attack: besides the price of reaching the staking cap itself, the attacker would need to pay the fee-price of having to fraction the attack into a high number of transactions due to the maximum stake amount per transaction.

The Babylon team has provided the following examples:

```
For example:
Staking Cap 5000 BTC, maximum stake amount 50BTC: cap can be filled
with minimum 100 transactions
Staking Cap 2000 BTC, maximum stake amount 0.01BTC: cap can be filled
with minimum 200000 transactions.

The larger the minimum amount of transactions, the more difficult this
attack becomes as Bitcoin block space is limited and expensive.
For example, in the last case, the attacker would have to entirely
capture 50 Bitcoin blocks (assuming 4k transactions per block), spend
14 BTC for fees (assuming staking tx size 200vbytes and the current
mainnet fee rate of 35sat/vbyte)
```

The calculations provided by Babylon are accurate. The likelihood of this issue depends entirely on the chosen parameters for the Phase 1. As stated in the assessment, the exact parameters to be used were not defined at the time of the review.

It is worth noting that if the attacker is a miner or a pool the fee-cost might be lower, as they are able to recover a part of the fees. It is also worth mentioning that even if the attacker does not reach the staking cap, having a high-enough amount of stake and then unbonding all of it could also have repercussions on the economic security of the the Babylon blockchain when it is deployed.

6. Disclaimer

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

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

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

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

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