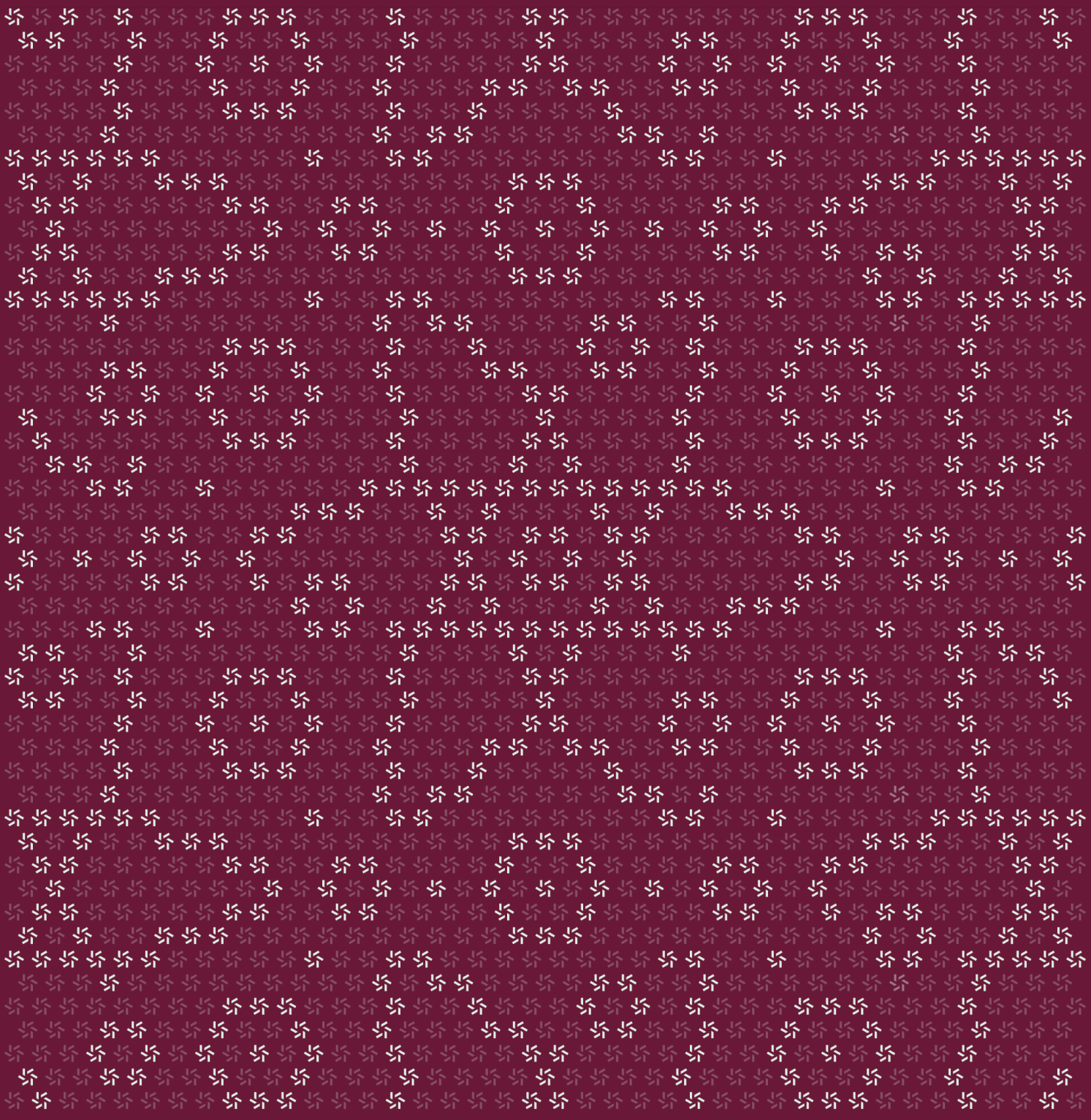
Zellic

June 28, 2024

Prepared for
Babylon

Prepared by
Syed Faraz Abrar
Jisub Kim
Ulrich Myhre
Avraham Weinstock
Zellic

Babylon
Blockchain Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Babylon from April 1st to May 31st, 2024. During this engagement, Zellic reviewed Babylon's code for security vulnerabilities, design issues, and general weaknesses in security posture.

Babylon plans to deploy the system in two phases. The first phase (Phase-1: Lock Only Network) involves only Bitcoin locking, where Bitcoin holders submit a staking transaction to the Bitcoin network. Slashing transactions will not be signed in this phase, so slashing will not be possible.

The Babylon mainnet phase-2 (Phase-2: Bitcoin Secured Babylon PoS chain) builds on phase-1 by adding support for Proof-of-Stake (PoS) systems, including the Babylon chain, to activate staking.

Our review covered both phases of their system; however, only phase-1 is included in this report. Since Babylon expects to make significant changes to the phase-2 system in the coming months, the report for those issues will be released at a future date.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Do the staking and unbonding transactions function as intended?
- Are signatures used correctly?
- Are the exposed RPC endpoints reasonable?
- Is the correct information passed between various components of the system?
- Is there a possibility for a malicious staker to avoid being slashed?
- Could a finality provider selectively slash one of their delegators without consequence?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Infrastructure relating to the project
- Key custody
- Front-end components, aside from the staking dashboard

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

Our engineers have invested a considerable amount of time gaining a deep understanding of the

codebase and all the components involved in both Phase 1 and Phase 2. However, since the majority of our efforts were focused on the Phase 1 components, we strongly recommend a re-audit of the Phase 2 components before they go live.

Additionally, the Babylon team has informed us that there will be several core changes and new functionalities implemented in the Phase 2 components in the near future.

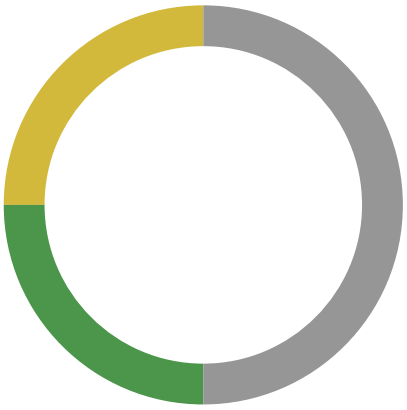
1.4. Results

During our assessment on the scoped Babylon modules, we discovered four findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Babylon's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	2



2. Introduction

2.1. About Babylon

Babylon contributed the following description of Babylon:

> Babylon ↗ is a shared security project whose vision is to leverage the security of Bitcoin for enhancing the security of PoS chains and L2s.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

Business logic errors. Business logic is the heart of any blockchain application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like stake getting stuck, or denial of service of critical components. To the best of our abilities, time permitting, we also review the blockchain logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of a component's interaction with other components. Time permitting, we review external interactions and summarize the associated risks.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Babylon Modules

Repositories	https://github.com/babylonchain/babylon/ ↗ https://github.com/babylonchain/btc-staking-ts ↗ https://github.com/babylonchain/simple-staking ↗ https://github.com/babylonchain/btc-staker ↗ https://github.com/babylonchain/cli-tools ↗ https://github.com/babylonchain/covenant-signer ↗ https://github.com/babylonchain/staking-api-service ↗ https://github.com/babylonchain/staking-indexer ↗ https://github.com/babylonchain/staking-expiry-checker ↗ https://github.com/babylonchain/staking-queue-client ↗
Versions	babylon: `8d76979ef05742206a74166c480c69b44b082798` btc-staking-ts: `6494df2b9f2c7a80578356659b1d24302e69dda2` simple-staking: `9040c942d0b811e880d284a69d8abbca0572614f` btc-staker: `5b60b53074c8bc49132b25594248becc23fd6e41` cli-tools: `d3921efd97bed74dbe9a3b8b578ab320e3460a52` covenant-signer: `91e4744bbe0bb440344354e380959d8126d9b82b` staking-api-service: `4e6033a0860df23400611bad24ec72934545f374` staking-indexer: `c13b4f0dd1a57f5f327e5fee613bd41e1b923062` staking-expiry-checker: `c04e2af4b38e363554b4a4b28485d484b837dbe3` staking-queue-client: `3f07eacc102a7ea9861689a4028c825d4a67e854`
Programs	• BTC Staker • BTC Staking Dashboard (simple-staking) • Babylon BTC Staking Library • Covenant Signer • Faucet • Staking API • Staking Expiry Tracker • Staking Indexer • Staking Queue Client • Unbonding Pipeline
Types	Go, TypeScript
Platform	Cosmos

2.4. Project Overview

Zellic was contracted to perform a security assessment with four consultants for a total of 25.7 person-weeks. The assessment was conducted over the course of two calendar months.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar
Engineer
faith@zellic.io

Jisub Kim
Engineer
jisub@zellic.io

Ulrich Myhre
Engineer
ulrich@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

April 1, 2024	Start of primary review period
April 4, 2024	Kick-off call
May 31, 2024	End of primary review period

3. Detailed Findings

3.1. Lack of input validations

Target	BTC Staking TS and Go		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The code does not properly validate user inputs in several areas:

- Numeric values
- Index values' range
- Length of buffer object

Additionally, required input values can be missing. The `stakingTimeLock` is two bytes for staking time, which can overflow when the input exceeds 65535 (approximately 455 days). Although there is a check in `validate()`, it is not utilized in the codebase.

```
buildDataEmbedScript(): Buffer {
  // 4 bytes for magic bytes
  const magicBytes = Buffer.from("01020304", "hex");
  // 1 byte for version
  const version = Buffer.alloc(1);
  version.writeUInt8(0);
  // 2 bytes for staking time
!  const stakingTimeLock = Buffer.alloc(2); // here
  // [...]
}
```

That `lockTime` is only two bytes also affects the Go implementation of `btcstaking`:

```
func buildTimeLockScript(
    pubKey *btcec.PublicKey,
    lockTime uint16,
) ([]byte, error) {
    builder := txscript.NewScriptBuilder()
    builder.AddData(schnorr.SerializePubKey(pubKey))
    builder.AddOp(txscript.OP_CHECKSIGVERIFY)
    builder.AddInt64(int64(lockTime))
    builder.AddOp(txscript.OP_CHECKSEQUENCEVERIFY)
    return builder.Script()
```

```
}
```

Impact

The lack of input validation can lead to unintended behavior or unexpected interruptions in code execution.

Recommendations

We recommend the following:

- Check that numeric values such as the amount, fee, and rates are nonnegative.
- Check that index values are within a valid range.
- Check and ensure that buffer objects have expected lengths, like checking `pks.length` in `utils/stakingScript.ts::buildMultiKeyScript`.
- Check that the required input values are not missing when creating an instance of the `StakingScriptData` class.
- Validate the script in the constructor.
- Use at least 32 bits for lock time (Assuming 10-minute blocks, there are 144 blocks per day, so 2^{32} blocks would not overflow for approximately 81,715 years).

Remediation

Babylon acknowledged this issue and created a fix in pull request #17 ↗.

3.2. Invalid creation of unbonding TX leads to loss of gas

Target	Simple Staking		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

> **Disclaimer**: The Babylon team had already fixed this issue approximately five hours prior to us reporting it.

In Babylon's staking dashboard, users are able to execute staking and unbonding transactions.

The unbonding transaction specifically must adhere to the following criteria:

1. It contains exactly one input, which points to the staking transaction's taproot output.
2. It contains exactly one output, which must be a taproot output committing to the BTC unbonding scripts recognized by Babylon.

It is important to note that the staking transaction can contain an arbitrary number of outputs, which means that the staking transaction's output index can be an arbitrary positive number.

When the staking dashboard generates the unbonding transaction, it does it correctly. The one and only output of the unbonding transaction will be the taproot output at index zero.

After the unbonding timelock period has expired, if the user wishes to withdraw their BTC, a withdrawal transaction must be broadcasted.

This withdrawal transaction has one input, which points to the taproot output of the unbonding transaction. In this case, the taproot output index will always be zero.

However, in the staking dashboard, the code actually passes in the staking transaction's output index to the `withdrawEarlyUnbondedTransaction()` function:

```
if (delegation?.unbondingTx) {
  // Withdraw funds from an unbonding transaction that was submitted for early
    unbonding and the unbonding period has passed
  withdrawPsbtTxResult = withdrawEarlyUnbondedTransaction(
    {
      unbondingTimelockScript,
      slashingScript,
```

```
    },
    Transaction.fromHex(delegation.unbondingTx.txHex),
    address,
    btcWalletNetwork,
    fees.fastestFee,
    delegation.stakingTx.outputIndex, // Incorrect index
  );
}
```

Impact

The impact of this is that users will be unable to unbond if their staking transaction's taproot output index was not set to zero.

We are not certain what the likelihood of this occurring is, but we assume a medium likelihood because the staking transaction's output index is not enforced or guaranteed to be 0.

Since users can still technically generate their own withdrawal transaction, we also assume a medium severity. It is important to note that typical users (i.e., the majority) will not have the technical capability to do this.

With a medium likelihood and medium severity, we determined a medium impact to the user as they are unable to withdraw their unbonded BTC.

Recommendations

Modify the call to `withdrawEarlyUnbondedTransaction()` in order to pass 0 as the output index.

Additionally, the call to `withdrawTimelockUnbondedTransaction()` must also be modified to pass in `delegation.stakingTx.outputIndex`. This code is in the same function.

Remediation

This issue has been acknowledged by Babylon, and a fix was implemented in commit 5bb21090 ↗.

3.3. Use alternative lib

Target	btc-staking-ts/src/utils/curve.ts		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

In `btc-staking-ts` library, it uses `ecc` from `@bitcoinerlab/secp256k1`, which is not implemented from `bitcoinjs`.

Impact

There is no guarantee that the `bitcoinerlab` will keep up with any interface changes in the future.

Recommendations

Use `@bitcoinjs-lib/tiny-secp256k1-asmjs`, which converts to a native JS from its maintainer.

Remediation

This issue has been acknowledged by Babylon, and a fix was implemented in commit 627d82c2 ↗.

3.4. Potential issues with the MinUnbondingTime parameter

Target	Staking Indexer		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Babylon chain has a `MinUnbondingTime` parameter, which specifies the minimum unbonding time that valid unbonding transactions must use in their time-lock script.

The staking indexer component tracks unbonding transactions by comparing the unbonding time in the time-lock script to an `UnbondingTime` parameter. Note that this `UnbondingTime` parameter is different to the `MinUnbondingTime` parameter in Babylon.

The issue is that the staking indexer performs an equality check (i.e., the unbonding time in the transaction must be equal to the `UnbondingTime` parameter in the indexer). Babylon, however, performs a greater-than-or-equal-to check (i.e., the unbonding time must be greater than or equal to `MinUnbondingTime`).

Impact

This leads to an issue where a user might create an unbonding transaction with the unbonding time set to a value greater than `MinUnbondingTime` but also not equal to the indexer's `UnbondingTime`. This would prevent the indexer from ever picking up the transaction, which means the covenant would never become aware of this transaction.

The impact from this is that the user would end up losing the gas fee that they paid for the unbonding transaction and subsequently would need to recreate it with the correct unbonding time.

Recommendations

Our initial recommendation for a fix was to modify the checks in Babylon such that the unbonding time in the transaction would have to match an exactly set parameter.

However, a finality provider can abuse this. The way finality providers can provide trust to users is by having one big self-delegation to themselves. This tells the users that the finality provider will not be malicious because they have the most to lose if they get slashed.

Now, if everyone has the same unbonding time, then the finality provider can unbond before any of the delegators. Once the unbonding transaction has gone through the time-lock period, the finality provider can withdraw and subsequently selectively slash any delegator they want without conse-

quence. This means that there must be some leeway to set the unbonding time.

Our current recommendation is for the checks in the staking indexer (and other components) be modified to be greater-than-or-equal-to checks, similar to how it is in Babylon.

Remediation

Babylon has informed us that they will leave the unbonding time checks as they are for now, since the Babylon chain has not been implemented yet. They additionally provided the following context for further clarity;

> We find this a non-issue, as the lock-only system is intended to have more strict unbonding time rules, requiring an exact unbonding time and not an unbonding time above a minimum. The staking-indexer utilizes the lock-only system rules as it is only intended to be used for this system. For phase-2, staking transactions are verified by the Babylon blockchain, which applies the phase-2 unbonding rules.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Overflowed transaction promotion

Automatically promoting the oldest overflow transaction that fits within the staking cap to active status would make sense when the amount of active stake decreases due to time-lock expiration or unbonding. While the Babylon team found this approach reasonable, they decided to keep the system simple. The goal of the cap is to ensure a certain amount of Bitcoin remains locked in the system, with any excess being unlocked.

4.2. Mitigating potential attacks through maximum staking value

Concerns were raised about the possibility of a single staker representing 100% of the BTC stake on Babylon if the first staking transaction stakes the entire cap (without overflowing). This could potentially allow them to attack proof-of-stake (POS) chains that honor Babylon stake in a way that would not be possible without the cap. The team addressed two points in response to this concern.

First, to prevent the scenario where a single staker can dominate the BTC stake on Babylon, the maximum staking value will be set much smaller than the overall staking cap.

Second, in Phase 1 of the Babylon protocol, there is no POS chain that could be attacked. The system parameters are designed to be upgradable, allowing for the staking cap to be increased in the future to accommodate more participants.

4.3. Improving front-end security

The Babylon team was concerned about front-end attacks such as CDN hacks and BGP hijack, and we discussed the appropriate security mechanisms for that.

XSS

The likelihood of XSS (cross-site scripting) in the current system appears to be low. Firstly, there are very few front-end pages, which reduces the attack surface. Secondly, most DOM elements are not easily controllable by users, making it harder to inject malicious scripts. Lastly, the use of Next.js makes it difficult for rendered values to be recognized as HTML tags, providing an additional layer of protection.

However, if an XSS vulnerability were to occur, it could have severe consequences, particularly in relation to the connected OKX wallet. The OKX wallet signs transactions without validating if the transaction data is related to staking. In the event of an XSS attack, an attacker could modify the API endpoint or pollute the transaction data, redirecting the user's UTXO to their own address, conducting an arbitrary transfer of funds.

The primary defense against XSS vulnerabilities would be to exercise caution and follow best practices when adding new code to the staking dashboard or other front-end components interacting with browser extensions like the OKX wallet. This includes validating and sanitizing user inputs, encoding output properly, implementing Content Security Policy (CSP), and regularly updating dependencies to ensure they are free from known vulnerabilities.

CDN hacks

The dashboard is not vulnerable to CDN hacks because it does not load any scripts from external resources.

At the application level, using nonexternal modules and subresource integrity (SRI) could be a way to improve security at the front end. SRI hash calculation is typically performed during the web-application development and deployment stages. Developers calculate the hash values of external resources in advance and include them in the integrity attribute of the HTML. Therefore, the hash values are already set before hacks occur.

BGP hijacking

From the perspective of defending BGP hijacks, RPKI (Resource Public Key Infrastructure) can be a way that helps validate the legitimacy of BGP route announcements. It ensures that the entity announcing a specific IP-address range is indeed authorized to do so. By using a service provider with RPKI implemented, such as Cloudflare or Akamai, they can benefit from an additional layer of protection at the network level more easily. For example, `balancer.fi` experienced BGP hijacking before, but Cloudflare was able to issue a warning ↗.

Since BGP hijacking is an issue at the network-infrastructure level, it is difficult for the main website itself to completely prevent BGP hijacking. In addition to applying RPKI, consistent monitoring should be conducted to ensure that the routing and AS are correct, as well as the integrity of the main page and third-party inclusions, to prevent abuse of the main site.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Transaction generation and signing

Specification

Transaction types

Babylon makes use of two transaction-output types: staking outputs and unbonding outputs.

1. Staking outputs commit to a taproot disjunction of three possible execution paths.
   - The timelock path, which requires the staker's signature and also requires a certain number of blocks to pass.
   - The unbonding path, which requires the staker's signature as well as a threshold of covenant emulation committee signatures.
   - The slashing path, which requires the staker's signature, the finality provider's signature, and a threshold of covenant emulation committee signatures.
2. Unbonding outputs commit to a taproot disjunction of two possible execution paths.
   - The timelock path, similar to the staking output's (but with a shorter duration in blocks than the staking output's timelock path).
   - The slashing path, with identical requirements to the staking output's slashing path.

Miniscript formulations

The different taproot script paths can be described in the miniscript language as follows:

- Timelock path: `and_v(vc:pk_k(staker_pk), older(timelock_blocks))`
- Unbonding path: `and_v(vc:pk_k(staker_pk), multi_a(covenant_threshold, covenant_pk1, ..., covenant_pkn))`
- Slashing path: `and_v(vc:pk_k(staker_pk), and_v(vc:pk_k(finalityprovider_pk), multi_a(covenant_threshold, covenant_pk1, ..., covenant_pkn)))`

Security properties

The staking and unbonding transactions, along with their bitcoin scripts, have a few security properties. These are:

- The timelock paths ensure that the system is fail-safe, in the sense that if all Babylon infrastructure ceases operating, the staker can eventually reclaim their stake with just the Bitcoin network.
- The covenant emulation committee signatures on the slashing path allow the committee to enforce that slashing transactions have additional structure — that they send a specified percentage of the input to an unspendable burn address and the remainder to a change address controlled by the staker.
- The staker and covenant emulation committee signatures provide defense-in-depth against slashing occurring in Phase 1. Phase 1 signing flows do not require the slashing path to be presigned by the staker, and `covenant-signer` (the Phase 1 counterpart to `covenant-emulator`) only signs unbonding paths.
- The finality provider's key is used to sign proof-of-stake blocks using EOTS, which leak the key on equivocation (i.e., validator misbehavior). For Phase 2 in the future, stake will only be considered active once the corresponding slashing path has been presigned by the staker and covenant emulation committee, which allow anyone to submit the slashing transaction upon validator misbehavior.
- The covenant signatures on the slashing path are adaptor signatures encrypted towards each finality provider to enforce atomic slashing (i.e., that anyone can ensure that if at least one delegator to a finality provider is slashed, all delegators to that finality provider are slashed).

`babylon/btcstaking`

The `btcstaking` library contains methods that construct and sign transactions in accordance with the aforementioned specification.

- `newBabylonScriptPaths`, called by `BuildStakingInfo` (for the staking output) and `BuildUnbondingInfo` (for the unbonding output), builds all three paths.
- `buildTimeLockScript` builds the timelock path, with `lockTime` as a parameter (see Finding 3.1. ↗).
- `buildSingleKeySigScript` and `buildMultiSigScript` are used to build individual signature-checking scripts and m-of-n multi-sig–checking scripts.
- In `newBabylonScriptPaths`, `unbondingPathScript` requires the staker's signature (via `stakerKey`) and `covenantQuorum` of the covenant signatures (from the set `covenantKeys`).
- In `newBabylonScriptPaths`, `slashingPathScript` requires the staker's signature, one finality-provider signature (from a set of `fpKeys`), and `covenantQuorum` of the covenant signatures.
- `SignTxWithOneScriptSpendInputFromTapLeaf` signs Bitcoin transactions.
- `EncSignTxWithOneScriptSpendInputStrict` signs Bitcoin transactions as adaptor signatures (where either the signature or the encryption private key can be recovered from the other).
- `ValidateSlashingTx` ensures (among other properties) that a candidate slashing transaction has the expected slashing and change outputs.

- `BuildV0IdentifiableStakingOutputsAndTx` constructs an unsigned partial transaction with no inputs and a staking output and `OP_RETURN` output with metadata.
- `ParseV0StakingTx` validates that an existing transaction is a staking transaction.

btc-staking-ts

The `btc-staking-ts` library contains methods that construct staking, unbonding, and slashing and sign transactions in accordance with the aforementioned specification.

- `stakingTransaction` generates an unsigned BTC staking transaction in PSBT format. The output includes an unsigned PSBT with the staking script, change, and optional data embed script, along with the total transaction fee.
- `withdrawTimelockUnbondedTransaction` generates transactions to withdraw unbonded staking funds. They call the `withdrawalTransaction` function to create the actual transaction.
- `unbondingTransaction` generates a transaction that converts staking funds to an unbonding state. It just returns PBST, needs to be signed with `signTransaction` or `createWitness`, and signs transactions by using BIP-0332. The staking script allows users to, on-demand, unbond their locked stake before the staking-transaction time lock expires, subject to an unbonding period.
- `createWitness` creates a witness for use in an unbonding transaction in PSBT format as, apart from the staker's signature, it also needs a set of signatures from the covenant emulation committee. It combines the original witness data with covenant-related data to generate a new witness.
- `slashingTransaction` generates a transaction that sends a portion of the staking funds to a slashing address and returns the remainder to the user when slashing conditions are met. The output consists of two transactions: one sending a portion of the input funds (input * `slashing_rate`) to the slashing address and the other sending the remaining input funds minus fee (input * (1-`slashing_rate`) - fee) back to the user's address.
- `withdrawal` generates a transaction that consumes the staking output to withdraw funds.
- `buildSingleKeyScript` and `buildMultiKeyScript` allow us to reuse functionality for creating Bitcoin scripts for the unbonding script and the slashing script.
- `buildMultiKeyScript` uses BIP-0342 (Tapscript) to build multikey scripts. It validates whether provided keys are unique and the threshold is not greater than the number of keys. If there is only one key provided, it will return single-key–sig script. It checks that the key must be sorted and verifies there are no duplicates.

cli-tools

The `cli-tools` binary contains several commands relevant to transactions:

- `create-phase1-staking-tx`, which uses `btcstaking.BuildV0IdentifiableStakingOutputsA` to produce an unsigned partial staking transaction (to be completed with `bitcoind`'s `fundrawtransaction` and `signrawtransactionwithwallet` commands)

- `create-phase1-unbonding-request`, which creates and signs (with the staker's key) an unbonding transaction, given information on the corresponding staking transaction
- `create-phase1-withdaw-request`, which creates and signs (with the staker's key) a timelock path of either a staking or an unbonding transaction
- `run-unbonding-pipeline`, which retrieves unbonding transactions from a MongoDB instance populated by `staking-api-service` and sends them to `covenant-signer` to attach covenant emulation committee signatures.

simple-staking

The `simple-staking` is a front-end dApp for creating Bitcoin staking transactions. It integrates with a set of extension wallets satisfying its expected interface. It is hosted by Babylon and serves as a reference implementation for entities that want to set up their own staking website. This uses `btc-staking-ts` for constructing staking/unbonding transactions, signs through extension wallets imported, then submits to staking and withdrawal transactions to Bitcoin. It also submits any unbonding transactions to the `staking-api`.

btc-staker

The `staker-cli` command contains the following Phase 1 commands under the `transaction` subcommand:

- `check-phase1-staking-transaction` validates a staking transaction with `btcstaking.ParseV0StakingTx` and optionally checks that provided additional data matches.
- `create-phase1-staking-transaction` creates an unsigned partial staking transaction with `btcstaking.BuildV0IdentifiableStakingOutputsAndTx`, similarly to `cli tools create-phase1-staking-tx`.
- `create-phase1-unbonding-transaction` creates a BIP174 PSBT for an unbonding transaction, given the corresponding staking transaction to be used as input.
- `create-phase1-staking-transaction-json` is similar to `create-phase1-staking-transaction` but takes its parameters through a JSON file instead of via CLI arguments.

The `stakerd` service contains behavior relevant to Phase 2, including automatically synchronizing state between the Bitcoin and Babylon chains and signing slashing transactions for staking transactions that have been processed by Bitcoin but that are not yet recognized as delegations by Babylon.

covenant-signer

The `covenant-signer` service exposes a "`/v1/sign-unbonding-tx`" route that provides covenant emulation committee signatures for unbonding transactions whose parameters are in bounds, whose only input is a staking output (that parses according to `btcstaking.ParseV0StakingTx`) that has sufficient confirmation depth on the Bitcoin blockchain, and whose only output is equal to an unbonding output reconstructed from the staking output's information. The `covenant-signer` service does not sign slashing transactions, which mitigates the risk of Phase 1 slashing.

staking-indexer

The `staking-indexer` service scans the Bitcoin blockchain for staking, unbonding, and withdrawal (i.e., timelock path) transactions that have the expected structure and that their values are in bounds (including checking statefully that the total stake does not exceed a parameterized cap) and forwards them as `{ActiveStakingEvent,UnbondingStakingEvent,WithdrawStakingEvent}`s to the queue.

staking-api-service

The `staking-api-service` service provides several HTTP GET endpoints that `simple-staking` uses to display the state of Babylon as well as a POST endpoint that is used to initiate unbonding (which saves the provided transactions and staker signature to the MongoDB table to be read by `unbonding-pipeline`). It also receives messages from the various queues, keeping its database state in sync with the Babylon and Bitcoin chains.

staking-expiry-checker

The `staking-expiry-checker` service connects to the same MongoDB instance as `staking-api-service`, which has a table of staking transactions together with the heights they will expire at. It retrieves transactions that have expired by the current height according to a Bitcoin client and submits `ExpiredStakingEvent`s to the queue for them, removing them from the database upon successful queue submission.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the canonical Bitcoin chain.

During our assessment on the scoped Babylon modules, we discovered four findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.