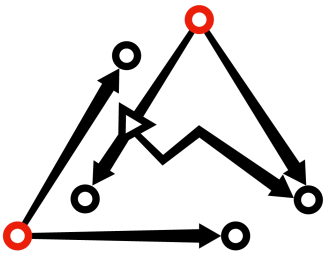

Least Authority

PRIVACY MATTERS

zkBTC Bridge Cryptography

Security Audit Report

zkBTC

FInal Audit Report: 17 April 2025

Table of Contents

Overview

Background

Lightec Ltd. has requested that Least Authority perform a security audit of zkBTC, a ZKP-based bridge for enabling Bitcoin in the Ethereum ecosystem.

Project Dates

- **March 3, 2025 - March 17, 2025:** Initial Code Review *(Completed)*
- **March 19, 2025:** Delivery of Initial Audit Report *(Completed)*
- **April 14, 2025 - April 16, 2025:** Verification Review *(Completed)*
- **April 16, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor and Writer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the zkBTC followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in-scope for the review:

- Recursive Length Prefix (RLP) Library:
  - *RLPark-0.2.3.zip* sent via email on February 28, 2025, and including:
    - `ListRlpCheck` (and all circuit functions it needs)
    - `RotateLeft`
    - `utils/util.go`
    - Excluding:
      - http://github.com/lightec-xyz/chainark `v0.5.7`
      - http://github.com/lightec-xyz/common `v0.2.7`
- Merkle Patricia Trie (MPT) Library:
  - *gMPTark-0.2.5.zip* sent via email on February 28, 2025, and including:
    - `mpt.go`, `nibble.go`, `node.go`
    - `keccak/keccak256.go` `keccak/hash.go`
      - Excluding `sha3.go` which is replicated from gnark
    - Excluding:
      - http://github.com/lightec-xyz/chainark `v0.5.7`
      - http://github.com/lightec-xyz/common `v0.2.7`
- BLS12-381 G2 signature verification:
  - PR: https://github.com/Consensys/gnark/pull/1040
    - Excluding the `_test.go` files
- SGX Enclave:
  - *zkbtcSgxServer-audit.b0228.zip* sent via email on February 28, 2025
    - Excluding the ZKP verifier from gnark

- Plonk verifier implemented in Rust:
    - Repo: https://github.com/lightec-xyz/plonk_verifier_on_icp/tree/main/src/plonk_verifier_on_icp_backend

For the review, the code was provided via shared folders and cloned into the following repositories for use during the audit and as a reference in this report:

- https://github.com/LeastAuthority/lightec-gMPTark
- https://github.com/LeastAuthority/lightec-RLPark
- https://github.com/LeastAuthority/lightec-zkbtc-SgxServer
- https://github.com/LeastAuthority/lightec-plonk-verifier-on-icp

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website: https://www.zkbtc.money
- Documentation: https://lightec.gitbook.io/lightecxyz/zkbtc-bridge
- Security document: https://github.com/lightec-xyz/zkBTC-Security

In addition, this audit report references the following documents and links:

- A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge." *IACR Cryptology ePrint Archive*, 2019, [GWC19]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Adversarial actions and other attacks on the bridge;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial-of-service attacks and security exploits that would impact or disrupt execution of the bridge;
- Vulnerabilities within individual components and whether the interaction between the components is secure;
- Exposure of critical information during interaction with external libraries;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other methods of exploitation;
- Data privacy, data leaking, and information integrity;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

We reviewed several components of the zkBTC protocol, which is a bridge between Bitcoin and Ethereum, using techniques from zero-knowledge proofs. The bridge allows a Bitcoin user to mint a zkBTC token, use this token in the Ethereum ecosystem, and eventually redeem the Bitcoin from the zkBTC token. When a Bitcoin is transferred into a zkBTC token, a proof is generated using a Succinct Non-interactive Argument of Knowledge (SNARK), in particular the PLONK SNARK [GWC19]. The PLONK proof verifies that the transfer was executed correctly, with the relevant inputs and outputs, without revealing any sensitive information. This proof is generated offline and verified via an Ethereum smart contract. When the zkBTC token needs to be redeemed at a later stage, this token is destroyed through an Ethereum smart contract. A PLONK proof is then generated offline that proves the zkBTC tokens were destroyed appropriately. The proof validation is followed by paying the user through a UTXO containing the transferred Bitcoins. The PLONK proof for redemption is validated either by the Bitcoin network or in a tamper-proof container. Once the proof is deemed to be valid, the UTXO is signed through a 2-out-of-3 multisignature, which entails the following three entities: a smart contract on the OASIS Sapphire blockchain, ICP tECDSA, and an Intel SGX enclave.

This protocol consists of several underlying components, and we examined a subset of them during this audit. In particular, we reviewed two libraries–`RLPark` (Recursive Length Prefix) and `gMPT` (Merkle Patricia Trie)–that contain utility functions for checking conditions associated with a transaction belonging to a specific Ethereum block, based on certain key parameters. Additionally, we reviewed the PLONK verification protocol, BLS signature verification process as per the Ethereum light client protocol, and the key generation process along with clients' connection to the SGX server. Further details are provided in the following section.

System Design

Our team examined the design of the zkBTC and found that security has generally been taken into consideration.

We reviewed the `RLPark` library, which contains utility functions for checking the correct formatting of Ethereum transactions in Recursive Length Prefix (RLP) format. We reviewed the `gMPTark` library, which implements a Patricia Merkle Trie. The `gMPT` library is used to check membership proofs of an Ethereum transaction within a certain Ethereum block, that the transaction receipts belong to the same block, and that the transactions and their corresponding receipts have the same MPT key.

We identified several utility functions within the `RLP` and `MPT` libraries where the code panics when handling invalid inputs. This behavior could lead to frequent system crashes if an attacker deliberately provides such inputs. We recommend implementing proper error handling to allow the code to manage external input validation more gracefully (Suggestion 3).

We additionally examined the PLONK verification protocol implemented in Rust, designed to run within a Dfinity-based ICP canister. We checked for the correctness of a Fiat-Shamir transcript and did not identify any issues in this regard. However, our team encourages adding thorough documentation of the proof circuit statement to help prevent future attacks resulting from unintended changes to the proof (Suggestion 1).

In addition, we analyzed the pull request adding BLS signature verification on the `BLS12-381` curve to Consensys' gnark library. This included the `hashing-to-G2` functionality and the newly implemented `AddUnified` function. Edge cases, such as identity element checks and subgroup validation, were implemented correctly.

During our review, we also assessed the private key generation and management process within an SGX enclave along with the related TLS-based client/server connection for sending and receiving SGX-generated private keys. We identified a few issues in this area of investigation: the secret value that leads to the private key is printed in plaintext in a log file (Issue A), and when an enclave report is invalid due to a TCB-level error, the code ignores this error and continues execution without interruption (Issue C).

Furthermore, when creating a new client, the flag `InsecureSkipVerify` is set to `true`. This results in a TLS connection being established without certificate validation, potentially enabling a man-in-the-middle attack (Issue B). Given the security-critical nature of this component, particularly its role in handling the communication of sensitive private keys, we also recommend adding detailed documentation to further clarify its design and security implications (Suggestion 1).

Dependencies

Our team did not identify any security issues in the use of dependencies. The zkBTC team utilizes a variety of tools, which are considered valid with no identified security risks. `BLS12-381` is a well-known curve in the ecosystem, and gnark and arkworks are reliable frameworks for generating PLONK proofs. While the use of Intel's SGX poses some considerations, it remains a suitable choice for this type of operation.

Code Quality

We performed a manual review of the repositories in scope and found the codebases to be generally organized and well-written. The zkBTC team utilizes gnark for proof generation and follows commonly used practices in the Bitcoin and Ethereum ecosystem, such as BLS signatures and a script opcode to verify the validity of a multisig transaction. However, we identified unresolved TODOs, as well as unused code and packages, which we recommend resolving or removing (Suggestion 3, Suggestion 4, and Suggestion 5).

Tests

Our team found the test coverage of the repositories in scope to be insufficient. A robust test suite should include, at a minimum, unit tests and integration tests that cover both success and failure cases. This helps identify errors and bugs and protect against potential edge cases, which could lead to security-critical vulnerabilities or exploits. We recommend improving the overall test coverage (Suggestion 2).

Documentation and Code Comments

The project documentation provided for this security review was insufficient in describing the general architecture of the system, each of the components, and how those components interact with each other. We recommend improving the project documentation to include a high-level protocol overview, given the complexity of the bridge between Bitcoin and Ethereum. Additionally, we suggest providing a detailed architecture diagram and refining the system specification.

While the codebase includes some code comments describing the intended behavior of security-critical components and functions, our team identified numerous typographical errors in comments and function names, which we recommend correcting (Suggestion 3, Suggestion 4, and Suggestion 5).

Scope

The scope of the review was limited, as we assessed only specific components of the overall system. While this allowed us to focus on the correctness of key factors, other critical aspects remained beyond

the scope of this review. We recommend conducting a follow-up audit, particularly for the smart contract.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Exposure of Secret Through Logging	Resolved
Issue B: InsecureSkipVerify Set To True for TLSConfig	Resolved
Issue C: Verification Ignores an Invalid Report With a TCB Level Error	Resolved
Suggestion 1: Update Documentation	Implemented
Suggestion 2: Include Tests for Edge and Failure Cases	Implemented
Suggestion 3: Improve Error Handling and Avoid Using Panics	Not Implemented
Suggestion 4: Correct Typographical Errors in Function Names and Code Comments	Implemented
Suggestion 5: Use Functions With Appropriate Names	Implemented

Issue A: Exposure of Secret Through Logging

Location

`lightec-zkbtc-SgxServer/blob/main/keyAgent.go#L49`

Synopsis

The function `backupSecret` takes the `secret` variable as input and writes it into a log file.

Impact

High.

An attacker who gains access to the log file can read the secret value in plaintext. Since this secret value is directly used for private key generation, the attacker would be able to control one of the keys for 2-out-of-3 signature generation. This poses a direct risk of allowing an attacker to forge transactions of their choice.

Feasibility

Given access to the log file, the attack can be executed immediately since the secret value is stored in plaintext in the log file.

Severity

Medium.

This rating reflects the high impact but low feasibility, as obtaining access to the log file remains challenging.

Preconditions

The attacker must have access to the log file.

Remediation

We recommend refraining from printing the secret value in the log file.

Status

The zkBTC team has resolved this issue by removing the printing of the secret value.

Verification

Resolved.

Issue B: InsecureSkipVerify Set To True for TLSConfig

Location

`lightec-zkbtc-SgxServer/blob/main/client/client.go#L20`

Synopsis

Through the function `NewClient`, a new client is created with a TLS configuration where `InsecureSkipVerify` is set to `true`. This enables establishing a connection with a server without verifying its certificate.

Impact

Medium.

This rating is based on the issue enabling a man-in-the-middle attack by allowing a connection to be established with a malicious client.

Feasibility

This exploit is moderately feasible, as skipping certificate verification allows a malicious connection to be established.

Severity

Medium.

Remediation

We recommend setting `InsecureSkipVerify` to `false` and using a proper certificate pool. When establishing a connection, we further recommend checking that the client certificate verifies to `true`.

Status

The zkBTC team opted to use an exported certificate instead of setting InsecureSkipVerify to true, thereby establishing two-way TLS authentication and allowing the TLS server to verify the client certificate. The code change can be found here.

Verification
Resolved.

Issue C: Verification Ignores an Invalid Report With a TCB Level Error

Location
lightec-zkbtc-SgxServer/blob/main/enclave.go#L64-L73

Synopsis
If the report contains an invalid TCB level, the code only throws a warning and proceeds without interruption.

Impact
High.

This rating reflects the potential for an attacker to gain access to sensitive data through the submission of an invalid report.

Feasibility
This attack is moderately feasible.

Severity
Medium.

Remediation
Rather than ignoring an invalid TCB level, we recommend defining a set of acceptable TCB statuses and failing the verification if the report does not meet the required security level.

Status
The zkBTC team has updated the code to return an error report when the TCB status is invalid, in accordance with the TCB levels defined here.

Verification
Resolved.

Suggestions

Suggestion 1: Update Documentation

Location
Throughout the codebase.

Synopsis
We reviewed the existing documentation provided for this review, and while it was helpful, we recommend expanding it to improve clarity and offer a more complete understanding of the bridge protocol.

Mitigation

We recommend updating the documentation to include a complete architecture diagram and a full technical specification, including a detailed description of the algorithms used, a PLONK proof statement, and the client/server protocol for communicating the SGX-generated private keys.

Status

The documentation has been updated to include the SGX-related client/server protocol. Details on the protocol, architecture, and proof statement can be found here and here.

Verification

Implemented.

Suggestion 2: Include Tests for Edge and Failure Cases

Location

`lightec-gMPTark/main/mpt_test.go`

`lightec-RLPark/main/rlp_test.go`

Synopsis

While the `gMPTark` and `RLPark` libraries include sufficient tests for success and correctness cases, the codebase lacks tests for edge and failure cases.

Sufficient test coverage should include tests for success and failure cases (all possible branches), which helps identify potential edge cases, and protect against errors and bugs that may lead to vulnerabilities. A test suite that includes sufficient coverage of unit tests and integration tests adheres to development best practices. In addition, end-to-end testing is also recommended to assess whether the implementation behaves as intended.

Mitigation

We recommend implementing comprehensive unit test coverage, including edge and failure cases, to detect any implementation errors and verify that the implementation behaves as expected.

Status

The zkBTC team has added tests for edge and failure cases in the RLP library.

Verification

Implemented.

Suggestion 3: Improve Error Handling and Avoid Using Panics

Location

Examples (non-exhaustive):

`lightec-RLPark/blob/main/rlp.go#L47-48`

`lightec-RLPark/blob/main/utils/util.go#L54-55`

`lightec-RLPark/blob/main/utils/util.go#L14-15`

Synopsis

Several functions (as listed above) immediately call `panic` when input parameters do not satisfy expected conditions. Although this method of handling invalid inputs does not directly enable an attack, it allows an attacker controlling the inputs to trigger a system crash, potentially resulting in a denial-of-service attack.

Mitigation

Instead of panicking when validating external inputs, we recommend returning an error message for invalid inputs. These messages should be propagated to the caller, allowing the calling code to handle errors appropriately.

Status

The zkBTC team acknowledged this suggestion but chose not to implement it for the following reasons:

They indicated that, as they are not currently hosting proving as a service, they do not expect to encounter related denial-of-service (DoS) attacks. Furthermore, they noted that even if errors were propagated to the caller, there would be limited ability to recover from them in practice. The client also pointed out that the gnark library itself frequently handles invalid inputs by triggering a panic, which aligns with their current handling approach.

Verification

Not Implemented.

Suggestion 4: Correct Typographical Errors in Function Names and Code Comments

Location

`plonk_verifier_on_icp_backend/src/point.rs#L56`

`plonk_verifier_on_icp_backend/src/point.rs#L74`

`plonk_verifier_on_icp_backend/src/witness.rs#L7-L13`

Synopsis

The functions in `point.rs` are incorrectly named as `ganrk_commpressed_x_to_ark_commpressed_x` and `ark_g1_to_gnark_unompressed_bytes`. Additionally, the comment in `witness` contains a typographical error.

Mitigation

We recommend renaming the functions to `gnark_compressed_x_to_ark_compressed_x` and `ark_g1_to_gnark_uncompressed_bytes` respectively, and correcting the typographical error in the code comment.

Status

The zkBTC team has corrected all identified typographical errors.

Verification

Implemented.

Suggestion 5: Use Functions With Appropriate Names

Location

lightec-zkbtc-SgxServer/blob/main/crypto/keypair.go#L21

Synopsis

The function `keypair.go` internally calls the function `PrivKeyFromBytes` from the external library `btcsuite`. The function `PrivKeyFromBytes` is misnamed, as it returns both a private and public key rather than only a private key.

Mitigation

If possible, we recommend coordinating with the engineers of `btcsuite` to assign the function a more accurate name.

Status

The zkBTC team has requested renaming of the function to the engineers of `btcsuite` through this issue.

Verification

Implemented.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.