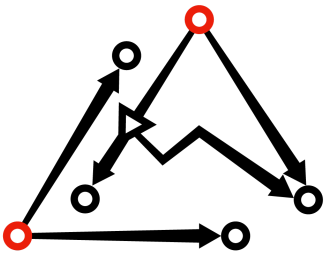

Least Authority

PRIVACY MATTERS

zkBTC Circuit and Smart Contracts

Security Audit Report

zkBTC

Final Audit Report: 16 June 2025

Table of Contents

This audit makes no statements or warranties and is for discussion purposes only.

Overview

Background

Lightec has requested that Least Authority perform a security audit of the zkBTC Circuit Implementation and Smart Contracts. The Circuit Implementation is for Ethereum and based on gnark, while the smart contract manages all zkBTC tokens, is ERC-20 compliant, and is ZKP-gated.

Project Dates

- **April 21, 2025 - May 23, 2025:** Initial Code Review *(Completed)*
- **May 27, 2025:** Delivery of Initial Audit Report *(Completed)*
- **June 16, 2025:** Verification Review *(Completed)*
- **June 16, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nikos Iliakis, Security Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the zkBTC Circuit Implementation and Smart Contracts 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:

- Circuit Implementation:
  https://github.com/lightec-xyz/provers
- Smart Contracts:
  https://github.com/lightec-xyz/zkBTC-contracts

Specifically, we examined the following Git revisions for our initial review:

- Circuit Implementation: `e016c3de22f37540cc489b72d479a282ca87439a`
- Smart Contracts: `f3b98c0ea67cd54061c13b09a2de2b380734e3ab`

For the verification, we examined the following Git revisions:

- Circuit Implementation: `bcee32fdcb81c3afa9b744998c62a65adab2a12d`
- Smart Contracts: `5333d24c842814362e1d0f3246fd3a9812d0cf71`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- Circuit Implementation:
  https://github.com/LeastAuthority/lightec-provers

This audit makes no statements or warranties and is for discussion purposes only.

- Smart Contracts:
  https://github.com/LeastAuthority/lightec-zkBTC-contracts

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-Security:
  https://github.com/lightec-xyz/zkBTC-Security

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution of the smart contracts and other components;
- Whether requests are passed correctly to the network core and between components;
- Key management, including secure private key storage and management of encryption and signing keys;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

zkBTC Bridge is a native zero-knowledge proof-based cross-chain protocol by Lightec that enables Bitcoin holders to convert BTC into an ERC-20 token, $zkBTC, on Ethereum at a 1 : 1 peg. Users lock BTC in the zkBTC bridge on Bitcoin and submit a succinct ZK proof to mint fully backed $zkBTC tokens on Ethereum, which can then be deployed across lending, trading, and yield-farming DeFi applications. When ready to reclaim their Bitcoin, holders burn their $zkBTC on Ethereum, generate a corresponding ZK proof of the burn, and unlock BTC on the Bitcoin network at parity. With its Beta V2 release, zkBTC Bridge also introduces a "Proving-as-Mining" mechanism, decentralizing proof generation by allowing any participant to mine ZK proofs in exchange for incentives. By applying ZKP techniques for transaction confidentiality

This audit makes no statements or warranties and is for discussion purposes only.

and to minimize on-chain verification costs, the zkBTC Bridge aims to enable users to access Ethereum's programmable finance.

System Design

Our team examined the zkBTC Circuit Implementation and Smart Contracts and found that it has been well-designed, with a strong focus on security. The sections below detail our observations and findings.

Smart Contracts

The smart contracts use strict role-based access control to restrict sensitive operations to authorized parties, and implement a checkpoint mechanism to verify that only well-confirmed Bitcoin transactions are accepted. Zero-knowledge proof verification is required for deposits and redemptions, and there are checks to prevent UTXO duplication and replay attacks. Additionally, the code uses assertions and `require` statements to enforce correct behavior and includes comments addressing potential denial-of-service risks.

As part of our review of smart contracts, we evaluated the protocol's UTXO management logic to assess susceptibility to double-spending and replay attacks, and found that while the contract enforces strict role-based access, the complexity of array handling and reliance on admin roles could present risks if not carefully managed.

We examined the Bitcoin transaction verification and checkpoint mechanisms for potential proof forgery or manipulation, and noted that security depends heavily on the correctness of the zero-knowledge proof integration and checkpoint rotation logic.

We also analyzed the contract's handling of large data structures and sorting operations for potential denial-of-service vectors, observing that gas exhaustion could be a concern if the UTXO set grows excessively.

Finally, we assessed migration and upgrade procedures for risks of state inconsistency or privilege escalation, and found that while access controls are in place, the migration process must be carefully monitored to prevent operational or security lapses.

Circuits

We reviewed the corresponding circuits for the zero-knowledge proof used within the system on the Ethereum side to burn the $zkBTC token (redeeming mechanism). The system is designed through five distinct circuits: `TxInEth2`, `BeaconHeader`, `BeaconHeaderFinality`, `Sync-Committee`, and the unifying circuit `Redeem`. The `TxInEth2` circuit checks for a block containing the `redeem` transaction. The `BeaconHeader` circuit checks for continuity between blocks, and the `BlockHeaderFinality` circuit checks for a correct signature by the sync committee for the latest block. The `Sync-Committee` circuit checks the continuity of the latest sync committee. The underlying zero-knowledge functionality is delivered through gnark and gnark-crypto with PLONK proofs over the BN254 curve.

In these circuits, gadgets from `gnark` are used, for example, a circuit representation of the MiMC hash to represent public keys. Additionally, the `provers` repository implements an SSZ gadget. Within these gadgets, we did not identify any issues. However, a Binary Merkle Tree gadget is implemented as well, where we identified two issues (Issue B, Issue C).

As part of our review, we also analyzed the circuits for missing constraints against their statement description in the README of the repository but did not identify any.

This audit makes no statements or warranties and is for discussion purposes only.

Dependencies

Our team examined the implemented dependencies in the codebase and identified no security issues in their use. For the smart contracts, the team utilizes the well-audited `OpenZeppelin` libraries, as well as actively maintained libraries from `OasisProtocol`. For the circuits, the project uses the audited Consensus gnark and gnark-crypto projects, and correctly applies SSZ, `MiMC`, `Keccak`, `SHA256` and both the `BLS12-381` curve (for signatures) and the `BN254` curve.

Code Quality

We performed a manual review of the in-scope repositories and found that the smart contract code is well organized and generally follows standard Solidity best practices, while the circuits implementation adheres to best practices for gnark usage. Functions are logically structured, access control is consistently enforced, and core operations are separated into clear modules. Additionally, the use of comments, `require` and `assert` statements, and established patterns such as role-based permissions and input validation are consistently applied.

Tests

The repositories in scope for the circuits are well-tested for correctness but lack failure tests. Additionally, while tests for the smart contracts exist, they rely on logs and manual inspection without incorporating assertions, which we suggest adding. We recommend improving the test suite for both the circuits and smart contracts (Suggestion 3).

Documentation and Code Comments

The project documentation provided by the Lightec team for the smart contracts was sufficient in describing the intended functionality of the system. For the circuits, the documentation was generally adequate and provided the necessary information to understand the complex circuit structure, although there were some typos and areas for improvement (Suggestion 1).

Additionally, for the smart contracts, comments were included in some cases to explain critical functionality, but the codebase could benefit from more comprehensive comments and clearer descriptions of function intentions. For the circuits, the code comments were sufficient, with function names appropriately selected and comments provided where necessary.

Scope

The scope of this review was sufficient for the smart contracts and included all security-critical components. However, our team noted that the `common` component of Lightec (`common/tree/master`) was not assessed as part of this review. This gap in coverage impacts the overall audit quality, as the fingerprint checking functionality is implemented within this component (Suggestion 2). Our team strongly recommends a comprehensive follow-up security audit focused on this area.

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: Loop Out of Bounds	Resolved

This audit makes no statements or warranties and is for discussion purposes only.

Issue B: No Domain Separation in Binary Merkle Tree	Resolved
Issue C: Possible Zero Merkle Tree Root Propagation in Circuits	Resolved
Suggestion 1: Improve Documentation	Implemented
Suggestion 2: Audit Fingerprinting Functionality	Implemented
Suggestion 3: Improve Test Suite	Implemented

Issue A: Loop Out of Bounds

Location

`contracts/deposit-redeem/zkBTCBridge.sol#L148`

Synopsis

The array loop will revert if a result is not found, due to arithmetic overflow.

Impact

Low.

The observed behavior results from a bug, not a security issue.

Feasibility

Medium.

Severity

Low.

Preconditions

This issue occurs when a search for an item in the array that does not exist is performed.

Technical Details

The loop termination condition is incorrectly defined. It should reversely iterate all elements until element zero, but the condition `(>=0)` will attempt to find a subsequent element, which, due to Solidity's overflow checks, results in a revert.

Remediation

We recommend modifying the logic such that the loop decrements the index only for elements greater than zero.

Status

The Lightec team has fixed the loop.

This audit makes no statements or warranties and is for discussion purposes only.

Verification

Resolved.

Issue B: No Domain Separation in Binary Merkle Tree

Location

circuits/types/binary_merkle_tree.go#L13-L26

circuits/utils/merkle.go#L50-L64

Synopsis

Due to the lack of domain separation, it is possible to forge a Binary Merkle Tree for a Binary Merkle Tree root.

Impact

High.

This issue could result in forged proofs for a Binary Merkle Tree without all data being verified. This undermines the integrity guarantees of the Binary Merkle Tree, which is being utilized in the `BeaconHeader` and `Sync-Committee Update` circuits.

Feasibility

Medium.

Severity

High.

Preconditions

The attacker must be able to provide the `leaves` input.

Technical Details

Within the code of the `BuildBinaryMerkleTree` function in `binary_merkle_tree.go`, the Binary Merkle Tree is constructed through this `for loop`:

```
Go
for i := l - 1; i > 0; i-- {
                tree[i] = sha256.Sum256(append(tree[2*i][:], tree[2*i+1][:]...))
        }
```

(Similar to, for example, in `merkle.go`).

Conceptually, this results in the leaves with hash values `h0=SHA256(l0)` and `h1=SHA256(l1)` (and data `l0` and `l1`) being hashed together into the internal node `n1=SHA256(h0||h1)`, and the leaves with hash values `h2=SHA256(l2)` and `h3=SHA256(l3)` (and data `l0` and `l1`) being hashed together into `n2=SHA256(h2||h3)`. In this simple example, the Merkle tree root would then be `r=SHA256(n1||n2)`.

This audit makes no statements or warranties and is for discussion purposes only.

However, this leads to a possible attack where the attacker can find a second preimage to the same Merkle tree root `r`: The tree consisting of only 2 leaves with hash values `n1` and `n2` (and not all `h0`, ..., `h3` and respective `l0`, ..., `l3`) since `SHA256(n1||n2)=r`.

Remediation

We recommend prefixing leaves with one value (e.g., `0`), and internal nodes with another value (e.g., 1). Through this domain separation, it is not possible to generate a second preimage to a hash value within the tree. (Alternatively, two distinct hash functions could also be used instead of prefixing).

Status

The Lightec team stated that the functionality within `circuits/utils/merkle.go` will not be used and has been removed.

Verification

Resolved.

Issue C: Possible Zero Merkle Tree Root Propagation in Circuits

Location

`circuits/types/binary_merkle_tree.go#L13-L26`

Synopsis

If the `leaves` input to the function generating the Binary Merkle Tree is empty, the `SHA256` hashing functionality is not executed and a Merkle tree root of zero is computed.

Impact

Medium.

This issue could result in constraints being incorrectly managed within the circuits of `BeaconHeader` and `Sync-Committee Update` against this Merkle Tree root due to it being zero.

Feasibility

Medium.

Severity

Medium.

Preconditions

The attacker must be able to provide the `leaves` input, which is a realistic scenario since the attacker would have access to the data to be proven through the circuits.

Technical Details

In the `BuildBinaryMerkleTree` function, the value `l` is computed through gnark-crypto's `NextPowerOfTwo` function:

```go
l := int(ecc.NextPowerOfTwo(uint64(len(leaves))))
```

If `len(leaves) == 0`, `l` will be set to 1. This then results in a tree being initialized with a root of zero

This audit makes no statements or warranties and is for discussion purposes only.

without ever entering the `for` loop. The output value will be a slice of two elements, each with a value of zero. However, this is not intended behavior for supplying an empty slice for `leaves` and can result in further problems within the constraint system for this Merkle tree with root zero.

Remediation

We recommend adding an input check for the edge case of `leaves` being empty, as well as defining and documenting a canonical empty Merkle tree, if necessary, and constraining it within the circuits of `BeaconHeader` and `Sync-Committee Update`.

Status

The Lightec team stated that the functionality within `circuits/utils/merkle.go` will not be used and has been removed.

Verification

Resolved.

Suggestions

Suggestion 1: Improve Documentation

Location

`lightec-provers/README.md`

Synopsis

During the audit, our team discovered imprecisions in the documentation, which reduce the readability of the code and, thus, make reasoning about the security of the system more difficult. Below we list some of our observations and recommendations:

- Update the diagram in the Architecture section by removing the duplicate Block `m+1` and adding Block `m`;
- Correct typographical errors in text passages for clarity (e.g., “conner” to “corner”); and
- Add data from the Google Sheet to the unified documentation doc.

Mitigation

We recommend clarifying the documentation in the README.md file by addressing the items listed above.

Status

The Lightec team has implemented the changes in PR#108.

Verification

Implemented.

Suggestion 2: Audit Fingerprinting Functionality

Location

Examples (non-exhaustive):

`redeem/core/redeem.go#L73`

This audit makes no statements or warranties and is for discussion purposes only.

sync-committee/core/outer.go#L47

Synopsis

The fingerprinting functionality is used across various parts of the code to recursively verify a verifying key, such as in the `Redeem` and `BeaconHeader` circuits. However, this functionality is not included in the `provers` repository but is located in an extra repository called `common`.

Mitigation

We recommend performing an independent audit of `common`.

Status

The Lightec team stated that the `common` functionality underwent a third-party audit in March 2025.

Verification

Implemented.

Suggestion 3: Improve Test Suite

Synopsis

The circuit repositories are thoroughly tested for correct behavior but do not include any tests for handling invalid inputs as well as edge cases. Similarly, while there are tests for the smart contracts, they rely on log output and manual review instead of assertions.

Mitigation

For the circuits, we recommend adding tests that handle failing and edge cases. (Within the circuit part, adding failure case testing is not crucial and is primarily intended for development and debugging, as proofs on invalid data will not be verified cryptographically. Edge case testing, however, is necessary to confirm that the circuits are correctly constrained). Additionally, instead of relying on manual inspection of the outputs of the tests, we recommend improving the tests for the smart contracts by incorporating assertion-based checks.

Status

The Lightec team has added edge case tests in PR#112.

Verification

Implemented.

This audit makes no statements or warranties and is for discussion purposes only.

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.

This audit makes no statements or warranties and is for discussion purposes only.