BEOSIN
Web3 Security & Compliance

zkBTC

Smart Contract and Circuit Security Audit Report

No. 202503211351

Mar 21st, 2025

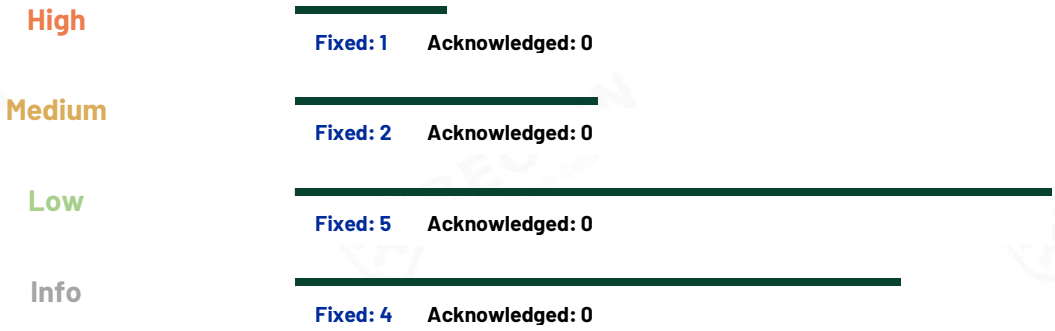
SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

After auditing, 1 High risk, 2 Medium-risk, 5 Low-risk and 4 Info items were identified in the zkBTC project. Specific audit details will be presented in the Findings for Circuit and Findings for Contract sections. Users should pay attention to the following aspects when interacting with this project：



1 Overview

1.1 Project Overview

Project Name	zkBTC
Project Language	Solidity and Go
Platform	Ethereum and Internet Computer
Code base	https://github.com/lightec-xyz/chainark https://github.com/lightec-xyz/common
FileHash(SHA256)	cc7f30b4832aa74e5c00e04de06796fa14d8d24093f3aa7c3cd1713305b0f69c 2843d0dde282789704ac8f5b310ce1ab2e6560e92814ee9de368d898495a6980 642298a669c85233db81b1e2d240239f48b3b7587cf2ba37c6e06ab3c41846db
Commit	acf76ccc5aa36a0e74ca477fda7d6317fd293bed 63de1a169d99600a7188ea37fcdd7a7ae03f12ec

1.2 Audit Overview

Audit work duration: Jan 12, 2024 – Mar 18, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's library of security expert rules. These rules call into the contracts and circuits under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

2. Manual Review

Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's and circuit's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's and circuit's business logic.

The manual audit is divided into three groups to cover the entire auditing process:

The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

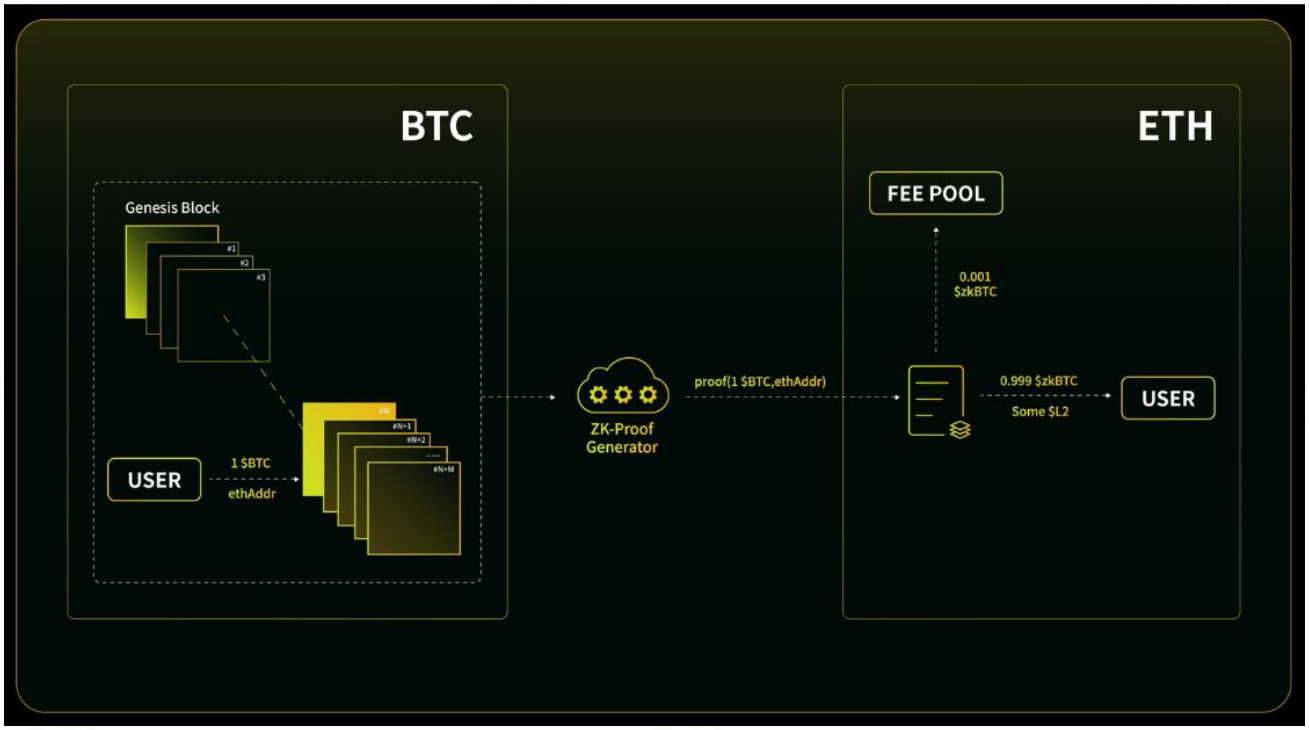
3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Project Description

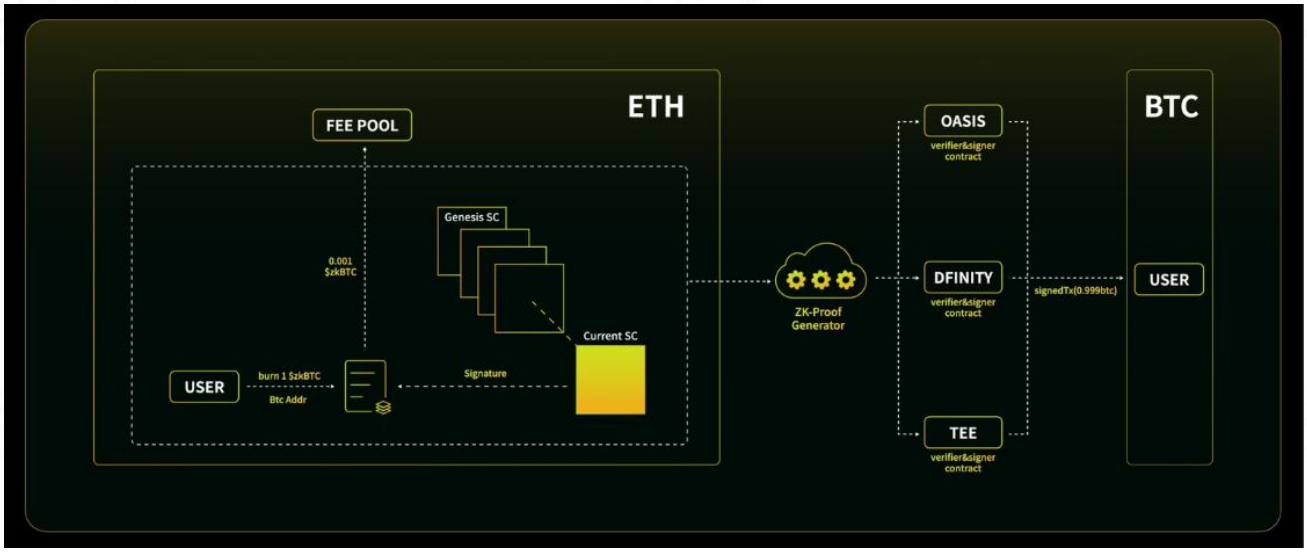
zkBTC is a blockchain project utilizing cross-chain bridge technology (zkBTCBridge) to enable bidirectional asset transfers between Bitcoin and Ethereum with zero-knowledge proof (zk) technology. Individuals can deposit BTC on the Bitcoin network, and after verification, an equivalent amount of zkBTC tokens along with additional L2 token rewards are minted on the Ethereum network. Conversely, individuals can burn zkBTC tokens to redeem their BTC, facilitating seamless cross-chain functionality with incentivized rewards. Below is a detailed introduction to the project:

❖ **Mint:**



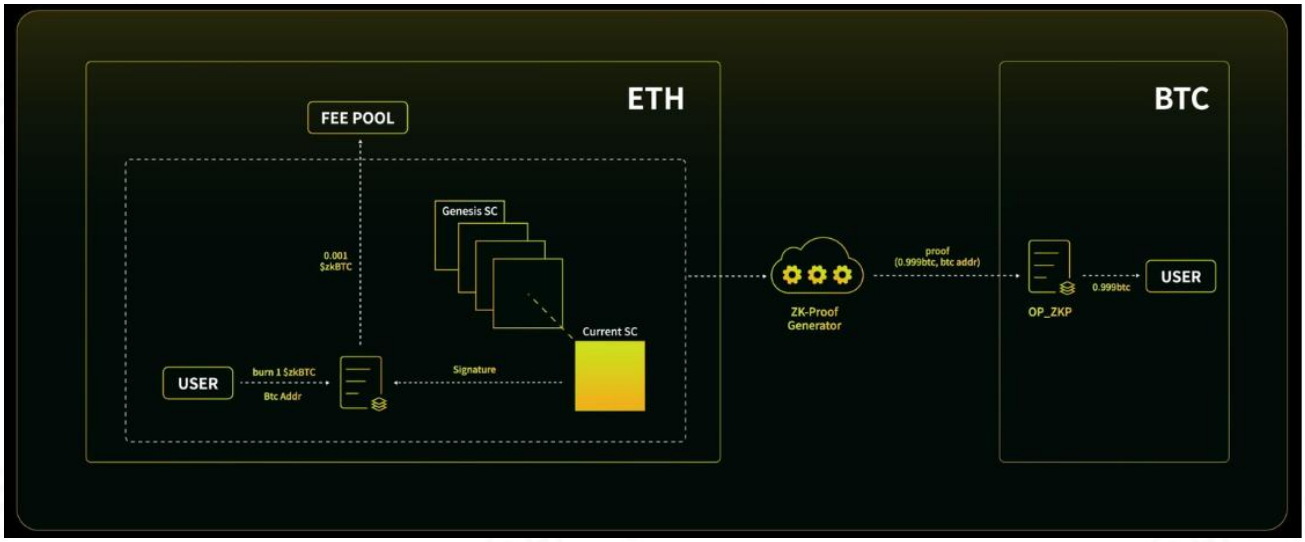
Individuals can mint $zkBTC by depositing $BTC to a designated Bitcoin address, where the deposit is linked to an Ethereum address (ethAddr) specified by the individual for receiving zkBTC. As shown on the BTC side of the diagram with a deposit of 1 $BTC from the individual, the ZK-Proof Generator creates an off-chain zero-knowledge proof (proof ($BTC, ethAddr)) to confirm the deposit, which is validated on the Ethereum network via a smart contract. The smart contract, on the ETH side, verifies the Bitcoin deposit transaction, ensuring it aligns with the blockchain from the Genesis Block to the block containing the deposit. It then mints 0.999 $zkBTC for the individual, with a 0.001 $zkBTC fee deducted and sent to the Fee Pool, maintaining a 1:1 peg between $BTC and $zkBTC.

❖ **Redeem :**



For the redemption process, an individual initiates the process by interacting with the Ethereum (ETH) smart contract, as shown on the ETH side of the diagram, where they submit 1 $zkBTC token to redeem their underlying $BTC, though the actual amount burned is 0.999 $zkBTC after fee deduction, as indicated in the diagram. This action triggers the ZK-Proof Generator to create an off-chain zero-knowledge proof, which verifies the validity of the redemption request. The proof is then submitted for verification, and depending on the implementation of the proposed opZKP opcode, two scenarios are possible: if opZKP is activated, the Bitcoin network directly verifies the proof, enabling the release of the corresponding UTXO (unspent transaction output) to the individual on the BTC side, as indicated by the "signed (0.999 $BTC)" flow to the individual, reflecting the deduction of a 0.001 $zkBTC fee prior to burning. If opZKP is not yet activated, the proof is verified in a tamper-proof container (labeled TEE, or Trusted Execution Environment) as an interim solution, with the verification process managed by entities like DFINITY and OASIS, which act as verifiers and signers of the blockchain contract. Once verified, the UTXO is spent, and the individual receives 0.999 $BTC on the Bitcoin network, as a 0.001 $zkBTC fee is deducted and sent to the Fee Pool on the ETH side before proof generation, maintaining the link between the burned $zkBTC and the redeemed $BTC. zkBTC relies on the Sync Committee from the Ethereum Light Client Protocol to ensure that the smart contract call was successful. As indicated in the diagram, there is a chain of certification from the Genesis Sync Committee to the Current Sync Committee to establish the validity of the Current SC.

❖ **When OP_ZKP is activated**



Once the OP_ZKP opcode is activated on the Bitcoin network, as indicated on the BTC side of the diagram, the redemption process becomes more streamlined. After an individual submits 1 $zkBTC on the Ethereum (ETH) side for redemption by interacting with the smart contract, where the SC refers to the Sync Committee in the Ethereum light client protocol (relying on its signatures to confirm Ethereum transaction finality in the design), the ZK-Proof Generator creates an off-chain zero-knowledge proof (proof (0.999 $zkBTC, btcAddr)) to confirm the validity of the redemption request. This proof is then submitted to the Bitcoin network, where the OP_ZKP opcode enables direct verification of the proof, as shown by the "OP_ZKP 0.999 $zkBTC" flow to the individual on the BTC side. Upon successful verification, the Bitcoin network authorizes the final payout of the corresponding UTXOs, releasing 0.999 $BTC to the individual's designated btcAddr. Meanwhile, on the ETH side, a 0.001 $zkBTC fee is deducted and sent to the Fee Pool prior to proof generation, maintaining the 1:1 peg between $BTC and $zkBTC throughout the process.

2.1 Circuit Description

btc_provers are responsible for generating Proofs that can demonstrate a user has indeed executed deposits and redemptions on the corresponding chain when BTC and ETH cannot directly interact, facilitating subsequent operations. The Circuit is divided into DepositTxCircuit and RedeemTxCircuit based on business logic.

Firstly, the DepositTxCircuit generates proof that a user has executed a valid deposit on BTC. This includes SigVerif, BlockDepths, TxInBlock, and BlockChain. SigVerif checks whether the latest block hash is correctly signed by the whitelisted entity DFinity, BlockDepths verifies that the transaction depth of the deposit and the CheckPoint depth meet expectations, TxInBlock checks whether the specified block contains the specified transaction, and BlockChain ensures that the current blockchain

used for calculations is valid. The transaction depth refers to the requirement that each deposit transaction must have a certain confirmation depth, while the CheckPoint depth mandates that the block containing the transaction is a descendant of certain known blocks.

Secondly, The RedeemTxCircuit is used to prove that a redemption transaction has been executed on Bitcoin, verifying the actual transfer. This proof enables the rewarding of miners who validate the Ethereum transaction and ensures that the change UTXO is returned under the management of the smart contract.

2.2 Contract Description

zkBTCBridge is the core component of the zkBTC project, utilizing zero-knowledge proof technology to enable bidirectional cross-chain transfers between the Bitcoin and Ethereum networks. Its primary functions are Deposit and Redeem: in the deposit process, users send BTC to a specified address, where the transaction's validity is confirmed, UTXO is recorded to prevent replay attacks, and fees are collected to distribute rewards. In the redemption process, users burn zkBTC tokens, prompting the selection of UTXO to generate an unsigned transaction; after the BTC transaction is completed, users submit proof to finalize redemption. The system handles UTXO and change management, validates BTC transactions, and operates a fee and reward distribution mechanism to incentivize L2 token holders, miners, and the foundation. It supports the minting and burning of zkBTC and LIT tokens, enforces permission controls for sensitive operations, accommodates various locking scripts, and sets minimum deposit amounts and transaction depth requirements to bolster security and prevent attacks.

3 Findings

Index	Risk description	Severity level	Status
zkBTC-Circuit-01	Inconsistency Between Definition and Instantiation in Deposit	Low	Fixed
zkBTC-Circuit-02	Unnecessary Circuit Complexity	Low	Fixed
zkBTC-Circuit-03	Incomplete Whitelist Verification for Verifier's VKey	Low	Fixed
zkBTC-Circuit-04	Issues in TxInMerkleProof with Fixed Depth Validation	Low	Fixed
zkBTC-Circuit-05	naming error	Info	Fixed
zkBTC-Contract-01	Incorrect random number generation mechanism	High	Fixed
zkBTC-Contract-02	Incorrect redeem function logic	Medium	Fixed
zkBTC-Contract-03	User Controllable Miner Fee	Medium	Fixed
zkBTC-Contract-04	The order of amount checks is unreasonable	Low	Fixed
zkBTC-Contract-05	Redundant code	Info	Fixed
zkBTC-Contract-06	Key functions are missing events	Info	Fixed
zkBTC-Contract-07	Function name is misspelled	Info	Fixed

Finding Details:

[zkBTC-Circuit-01] Inconsistency Between Definition and Instantiation in Deposit

Severity Level	Low
Lines	btc_provers-feat-gnark_v0.12\circuits\txinchain\deposit.go #48
Type	Business Security
Description	When defining the deposit circuit, whether to introduce the `SigVerif` component is determined by `common.SIGNED_TIP`. However, when instantiating the `NewDepositTxCircuit`, the component circuit is instantiated regardless of whether `SigVerif` is introduced or not. This may lead to inconsistencies between the definition and instantiation.
Recommendation	It is recommended to judge the `SIGNED_TIP` in the `NewDepositTxCircuit` function to decide to use `SigVerif`.
Status	**Fixed.** The code has added logic to determine `SIGNED_TIP` in the `NewDepositTxCircuit` function.

[zkBTC-Circuit-02] Unnecessary Circuit Complexity

Severity Level	Low
Lines	btc_provers-feat-gnark_v0.12\circuits\sigverif\sigverif.go #17
Type	Business Security
Description	During signature verification, sigverif splits R and S in the signature into two 16-byte elements instead of the typical 32-byte R and 32-byte S, introducing unnecessary complexity due to byte reversal.
Recommendation	It is recommended to use the standard 32-byte R and 32-byte S to simplify the byte order issue.
Status	**Fixed.**

[zkBTC-Circuit-03] Incomplete Whitelist Verification for Verifier's VKey

Severity Level	Low
Lines	chainark-main\verifier.go#26-53
Type	Business Security
Description	In the Define function of the `Verifier` circuit, there is a check to determine if the VKey belongs to the whitelist. However, the current implementation allows the verification to pass even if the VKey is not part of the `vkeyFpsBytes` whitelist and `vkeyFpsBytes` is not entirely present in the Witness.

```
func (c *Verifier[FR, G1El, G2El, GtEl]) Define(api frontend.API) error
{
    if c.NbSelfFps != len(c.VkeyFpsBytes) {
        panic("length mismatch")
    }
    vkeyFp, err := common_utils.InCircuitFingerPrint[FR, G1El,
G2El](api, &c.VKey)
    if err != nil {
        return err
    }
    vkeyFps := make([]common_utils.FingerPrint[FR],
len(c.VkeyFpsBytes))
    for i := 0; i < len(c.VkeyFpsBytes); i++ {
        vkeyFps[i] =
common_utils.FingerPrintFromBytes[FR](c.VkeyFpsBytes[i])
    }
    recursiveFpTest := common_utils.TestFpInFpSet[FR](api, vkeyFp,
vkeyFps)
    initialOffset := c.NbIdVars * 2
    nbFpVars := c.NbFpVars
    setTest := TestRecursiveFps[FR](api, c.Witness, vkeyFps,
initialOffset, nbFpVars, c.NbSelfFps)
    api.AssertIsEqual(recursiveFpTest, setTest)
    verifier, err := plonk.NewVerifier[FR, G1El, G2El, GtEl](api)
    if err != nil {
        return err
    }
    return verifier.AssertProof(c.VKey, c.Proof, c.Witness,
```

```
plonk.WithCompleteArithmetic())
}
```

Recommendation It is recommended to determine in the code that `setTest` is equal to 1.

Status **Fixed.** The constraint that `setTest` is equal to 1 has been added to the code.

```
initialOffset := c.NbIdVars * 2
nbFpVars := c.NbFpVars
setTest := TestRecursiveFps[FR](api, c.Witness, vkeyFps,
initialOffset, nbFpVars, c.NbSelfFps)
api.AssertIsEqual(1, setTest)
```

[zkBTC-Circuit-04] Issues in TxInMerkleProof with Fixed Depth Validation

Severity Level	Low
Lines	btc_provers-feat-gnark_v0.12\circuits\txinblock\circuit.go#37
Type	Business Security
Description	In the validation of TxIndex transactions within a block, a fixed `MaxMerkleTreeDepth` of 13 is enforced, necessitating 13 iterations regardless of the actual number of transactions present. This situation presents two key concerns: Low Transaction Count: When the number of transactions in a block is low, this inefficiency can lead to increased processing times and resource consumption. Potential for High Transaction Count: Recent block data from Bitcoin indicates that while the average number of transactions per block is around 2,000, the introduction of SegWit has removed the previous 1MB limit. This means that in extreme cases, the transaction count could exceed 8,192 (2^13). In n such extreme cases, the Merkle verification would fail, making proof generation impossible.
Recommendation	It is recommended to dynamically validate the Merkle tree using len(tp.MerklePath). For reference, here's an official example code from gnark: https://github.com/Consensys/gnark/blob/master/std/accumulator/merkle/verify.go.
Status	**Fixed.** The project team identified that a `MaxMerkleTreeDepth` of (2^13) is insufficient for extreme cases. They determined that (2^14) is required to handle such scenarios effectively. However, to accommodate potential future Bitcoin upgrades, they have decided to update the value to (2^16).

[zkBTC-Circuit-05] naming error

Severity Level	Info
Lines	blockchain\baselevel\circuit.go#27,32
Type	Coding Conventions

Description	The parameter `firstBlcokHash` in the circuit.go file is named incorrectly.
	`firstBlcokHash := types.Hash(firstBlockHashBytes)`
	`calcedHash, err := types.DoubleSha256(api, c.BlockHeaders[0][:])`
	`if err != nil {`
	`return err`
	`}`
	`firstBlcokHash.AssertIsEqual(api, *calcedHash)`

Recommendation	It is recommended that the `firstBlcokHash` parameter be changed to `firstBlockHash`.
Status	**Fixed.**

[zkBTC-Contract-01] Incorrect random number generation mechanism

Severity Level	High
Lines	contracts/verify-sign/BtcTxVerifier.sol #L119
Type	Business Security

Description	The `tryRotateCP` function uses the current block hash to generate a random number. However, according to Ethereum's mechanism, the block is not finalized when the transaction is executed, so the return value of uint256(`blockhash(block.number)`) will always be 0. Clearly, this is not a valid random number generation mechanism.
	`if (block.timestamp - old > ROTATION_THRESHOLD) { // time for a new CP`
	`if ((useTick && tockCandidates.length == 0) || (!useTick && tickCandidates.length == 0)) {`
	`// no candidates, skip this time`
	`return;`
	`}`
	`// pick a new CP from the candidates`
	`uint256 oracle = uint256(blockhash(block.number));`
	`bytes32 newCp;`

Recommendation	Due to the lack of a robust random number generation mechanism in Ethereum, it is recommended to choose a fix based on the specific business requirements. If high randomness is required, it is advisable to use a decentralized oracle service like Chainlink VRF (Verifiable Random Function) to generate secure and unpredictable random numbers. If the randomness requirement is not as high, blockhash(block.number - 1) can be used to obtain a random number.

Status	**Fixed.** Code has been modified to get oracle logic.
	`if (block.timestamp < ROTATION_THRESHOLD + seniorCP.timestamp // not time for a new CP`
	`|| cpCandidates.length == 0) { // or no candidates yet`
	`return;`
	`}`
	`// pick a new CP from the candidates`
	`uint256 oracle = uint256(keccak256(abi.encode(suggestedCP())));`

[zkBTC-Contract-02] Incorrect redeem function logic

Severity Level	**Medium**
Lines	contracts/deposit-redeem/FeePool.sol #L268-316
Type	Business Security
Description	In the `FeePool` contract within the deposit-redeem module, the `splitRedeemReward` function will be called by the `zkBTCBridge` contract to burn the zkBTC of the redeemer and distribute the redeem rewards. However, as shown in the code below, the logic in the `splitRedeemReward` function mistakenly treats msg.sender as the redeemer, resulting in incorrect deductions of the zkBTC token and improper reward distribution.
Recommendation	It is recommended to modify the msg.sender in the `splitRedeemReward` function to the original redeemer, i.e., the user who calls the redeem function and needs to redeem BTC.
Status	**Fixed.** Code has been modified by the caller.

```
    function splitRedeemReward(
        uint64 redeemAmount,
        uint64 btcMinerFee,
        uint64 targetAmount,
        uint64 feeAmount,
        address foundation
    ) external onlyRole(DEFAULT_ADMIN_ROLE) returns (uint256) {
        redeemBalanceCheck(msg.sender, redeemAmount + btcMinerFee);
        // postpone fee collection and redeem burning to after finding
UTXOs successfully
        zkBTCInterface.burn(msg.sender, targetAmount);
```

```
    function splitDepositReward(uint64 amount, address foundation,
address receiveAddress,
        address zkpMiner) external onlyRole(DEFAULT_ADMIN_ROLE){
        // mint zkBTC for user
        (uint64 userAmount, uint64 feeAmount) = getBridgeToll(amount);
        zkBTCInterface.mint(receiveAddress, userAmount);
```

[zkBTC-Contract-03] User Controllable Miner Fee

Severity Level	**Medium**
Lines	Contracts/deposit-redeem/zkBTCBridge.sol#L188
Type	Business Security
Description	In the redeem function, the user can control the value of `btcMinerFee`, if the user enters a value of zero for `btcMinerFee`, this will result in the user not having to pay the miner's fee.

```
    function redeem(uint64 redeemAmount, uint64 btcMinerFee, bytes
memory recipientLockScript)
    external checkLockScript(recipientLockScript) {
        if (recipientLockScript.equal(changeLockScript)) {
            revert LockScriptIsChange(recipientLockScript);
        }
        (uint64 userAmount, uint64 feeAmount) =
feeAccount.getBridgeRedeemToll(redeemAmount);
        uint64 targetAmount = uint64(userAmount) + btcMinerFee;
        (UTXOManagerInterface.UTXO[] memory spendUTXOs, uint64
totalAmount)
            = utxoManager.spendUTXOs(targetAmount);
        uint256 minerReward =
feeAccount.splitRedeemReward(redeemAmount, btcMinerFee, targetAmount,
feeAmount, foundation);
        createRedeemUnsignedTx(spendUTXOs, uint64(userAmount),
totalAmount-targetAmount, recipientLockScript, minerReward);
    }
```

Recommendation	It is recommended to check if the `btcMinerFee` entered by the user is reasonable.
Status	**Fixed.** An algorithm has been implemented in the code that automatically adjusts the minimum miner's fee.

[zkBTC-Contract-04] The order of amount checks is unreasonable

Severity Level	Low
Lines	Contracts/deposit-redeem/UTXOManager.sol#L92-101,L108-124, contracts/deposit-redeem/FeePool.sol #L93-87
Type	Business Security
Description	As shown in the following code, in this project's contract, the quantity limit is checked first and updated afterward. This sequence is unreasonable because it's possible that the check shows the limit has not been reached, but after the update, it could exceed the limit.

```
    function registerFee(uint64 fee) public
onlyRole(DEFAULT_ADMIN_ROLE) {
        require(fee < MAX_FEE_AMOUNT, "too much fee");
        require(
            totalzkBTCAmount < MAX_zkBTC_AMOUNT,
            "zkBTC amount exceeds possible max"
        );
        totalzkBTCAmount += fee;
    }
    function addUTXO(bytes32 txid, uint32 index, uint64 amount

    ) external checkUTXO(txid, amount) onlyRole(DEFAULT_ADMIN_ROLE) {

        assert(totalAvailableAmount < MAX_AMOUNT);

        utxos[txid] = UTXO(txid, index, amount, true);

        totalAvailableAmount += amount;

        recordShuntInfo(txid, amount);

        emit UTXOAdded(txid, index, amount);

    }

    function updateChange(bytes32 txid) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        UTXO memory utxo = utxos[txid];
        if (utxo.amount == 0) {
            revert ChangeNotExisting(txid);
        }
        if (utxo.isChangeConfirmed) {
```

```
            revert ChangeAlreadyUpdated(txid);
        }
        assert(totalAvailableAmount < MAX_AMOUNT);
```

Recommendation It is recommended to update first and then perform the limit check, or include the current amount in the check.

Status **Fixed.** The code has been optimized to update the data first, then perform the check.

```
    function _registerFee(uint64 fee) private{
        totalzkBTCAmount += fee;
        require(totalzkBTCAmount < MAX_zkBTC_AMOUNT, 'zkBTC amount
exceeds possible max');
    }
    function addUTXO(bytes32 txid, uint32 index, uint64 amount
    ) external checkUTXO(txid, amount) onlyRole(DEFAULT_ADMIN_ROLE) {
        utxos[txid] = UTXO(txid, index, amount, true);
        totalAvailableAmount += amount;
        assert(totalAvailableAmount < MAX_AMOUNT);
        recordShuntInfo(txid, amount);
        emit UTXOAdded(txid, index, amount);
    }
    function updateChange(bytes32 txid) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        UTXO memory utxo = utxos[txid];
        if (utxo.amount == 0) {
            revert ChangeNotExisting(txid);
        }
        if (utxo.isChangeConfirmed) {
            revert ChangeAlreadyUpdated(txid);
        }
        utxo.isChangeConfirmed = true;
        utxos[txid] = utxo;
        totalAvailableAmount += utxo.amount;
        assert(totalAvailableAmount < MAX_AMOUNT);
```

[zkBTC-Contract-05] Redundant code

Severity Level	Info
Lines	Contracts/deposit-redeem/zkBTCBridge.sol #L203
Type	Coding Conventions

Description	The zkBTCBridge contract declares the `RECEIVE_INDEX` variable but does not use it, making it redundant code.
	`uint8 constant RECEIVE_INDEX = 0;`
	`uint8 constant CHANGE_INDEX = 1;`

Recommendation	It is recommended to remove the unused redundant code.
Status	**Fixed.** Code logic has been modified to ensure that RECEIVE_INDEX is used.

```
    uint8 constant RECEIVE_INDEX = 0;
    uint8 constant CHANGE_INDEX = 1;
    function getTxOuts(uint64 userAmount, uint64 changeAmount, bytes
memory receiveLockScript, bytes memory changeLockScript)
    public pure returns (BtcTxLib.TxOut[] memory) {
        BtcTxLib.TxOut[] memory outputs = new BtcTxLib.TxOut[](1);
        if (changeAmount > 0) {
            outputs = new BtcTxLib.TxOut[](2);
            outputs[CHANGE_INDEX] = BtcTxLib.TxOut({
                value: changeAmount,
                pkScript: changeLockScript
            });
        }
        outputs[RECEIVE_INDEX] = BtcTxLib.TxOut({
            value: userAmount,
            pkScript: receiveLockScript
        });
        return outputs;
    }
```

[zkBTC-Contract-06] Key functions are missing events

Severity Level	Info
Lines	contracts/verify-sign/BtcTxVerifier.sol #L119
Type	Coding Conventions
Description	In the project contract, some functions that modify key variables do not emit events, which is not a good development practice.

```
    function updateFoundationAddress(address _foundationAddress)
external {
        require(foundation != address(0) && msg.sender == foundation,
'only current foundation can change its address');
        foundation = _foundationAddress;
    }
    function setFoundationAddress(address _foundationAddress) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        require(foundation == address(0), 'operator can only set
foundation if not set before');
        foundation = _foundationAddress;
    }
    function updateTxVerifier(IBtcTxVerifier _newVerifier) external
onlyRole(DEFAULT_ADMIN_ROLE){
      txVerifier = _newVerifier;
    }
    function updateUtxoAddress(UTXOManagerInterface _utxoAddress,
bytes memory _changeLockScript,
        bytes memory _multiSigScript) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        utxoManager = _utxoAddress;
        changeLockScript = _changeLockScript;
        multiSigScript = _multiSigScript;
    }
```

Recommendation	It is recommended to declare the corresponding events and trigger them within the functions.
Status	**Fixed.** The code has added event triggering.

[zkBTC-Contract-07] Function name is misspelled

Severity Level	Info
Lines	Contracts/verify-sign/BtcTxVerifier.soll #L81
Type	Coding Conventions

Description	The `updateCnadidateaThreshold` function is misnamed.
	`function updateCnadidateaThreshold(uint64 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {`
	`require(amount >= 10000000 && amount <= 1000000000, 'threshold must be within [0.1, 10]');`
	`candidateThresholdAmount = amount;`
	`}`

Recommendation	It is recommended to correct the function name to improve clarity and better reflect the function's purpose.
Status	**Fixed.** This function has been removed.

4 Appendix

4.1 Vulnerability Assessment Metrics and Status in Smart Contracts

4.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1(Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

4.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

4.1.3 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

4.1.4 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

4.2 Audit Categories

No.	Categories	Subitems
Contract	Coding Conventions	Deprecated Items
		Redundant Code
		require/assert Usage
		Default Values
	General Vulnerability	Insufficient Address Validation
		Lack Of Address Normalization
		Variable Override
		DoS (Denial Of Service)
		Function Call Permissions
		Call/Delegatecall Security
		Tx.origin Usage
		Returned Value Security
		Mathematical Risk
		Overriding Variables
	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Arbitrage Attack
		Access Control
Circuit	General Vulnerability	Assigned but Unconstrained
		Missing Input Constraints
		Unsafe Reuse of Circuit
		Arithmetic Field Errors
		Other Programming Errors
		Out-of-Circuit Computation Not Being Constrained
		Over-Constrained

	Business Security	Wrong translation of logic into constraints
		Bad Circuit/Protocol Design
		ZKP Complementary Logic Error
		Incorrect Custom Gates

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Additionally, the security issues of circuits were divided into two types: General Vulnerability and Business Security.

The specific definitions of contract issues are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

The specific definitions of circuit issues are as follows:

- **General Vulnerability**

General Vulnerability refers to a class of defects in arithmetic circuit design that arise from improper definition, implementation, or management of constraints. These issues can prevent the circuit from accurately and safely reflecting the intended logic. Examples include unconstrained variables or inputs (Assigned but Unconstrained, Missing Input Constraints), unsafe reuse of circuits (Unsafe Reuse of Circuit), arithmetic field errors (Arithmetic Field Errors), programming mistakes (Other Programming Errors), unverified out-of-circuit computations (Out-of-Circuit Computation Not Being Constrained), and excessive constraints (Over-Constrained). Such problems can lead to unexpected solutions, security failures, or functional errors, particularly compromising the credibility of proofs or exposing sensitive information in zero-knowledge proof systems.

- **Business Security**

Business Security pertains to issues that arise during the design and implementation of zero-knowledge proofs (ZKP) and arithmetic circuits due to errors in translating business logic into circuit constraints or flaws in protocol design. These problems can hinder the system's ability to achieve its intended business objectives or security assurances. Examples include incorrect translation of logic into constraints (Wrong Translation of Logic into Constraints), poorly designed circuits or protocols (Bad Circuit/Protocol Design), complementary logic errors within the ZKP system (ZKP Complementary Logic Error), and errors in the implementation of custom gate circuits (Incorrect Custom Gates).

* Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

4.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

4.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Web3 Security & Compliance

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

X
https://x.com/Beosin_com

Email
service@beosin.com

LinkedIn
https://www.linkedin.com/company/beosin/