BLOCKSEC

Security Audit

Report for atoshi-privacy-circuits

Date: July 31, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Atoshi
Target	atoshi-privacy-circuits

Version History

Version	Date	Description
1.0	July 31, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-17 03:28:02+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Circuit
Language	Circom
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of atoshi-privacy-circuits of Atoshi.

The `atoshi-privacy-circuits` project is a Circom/Groth16 zero-knowledge circuit system for shielded transactions. It contains three core circuits. `Shield` creates private note commitments for deposits. `Unshield` proves note ownership and enables withdrawals with nullifier-based double-spend protection. `Transfer` spends an existing note and creates a new note while preserving asset consistency.

Note this audit focuses on the code located in the following directories:

- `circuits/core`
- `circuits/lib`

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
atoshi-privacy-circuits	`Version 1`	`759fd4b2668e94bf23aad9e7bcb62e936e8ee2e7`
	`Version 2`	`c20bf8be5e622c4350b53cd3010263e1a4ff8424`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

[1] https://github.com/ATOSHI-ORG/atoshi-privacy-circuits

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (i.e., Circom), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **five** potential security issues. Besides, we have **three** recommendations and **two** notes.

- High Risk: 4
- Medium Risk: 0
- Low Risk: 1
- Recommendation: 3
- Note: 2

ID	Severity	Description	Category	Status
1	High	Compilation errors	Security Issue	Fixed
2	High	Incorrect range in template `AmountRangeCheck`	Security Issue	Fixed
3	High	Lack of binding on withdrawal signals	Security Issue	Fixed
4	High	Lack of necessary public inputs	Security Issue	Fixed
5	Low	Lack of range constraint for signal `fee`	Security Issue	Fixed
6	-	Remove redundant code	Recommendation	Fixed
7	-	Clarify `tokenId` semantics in note commitments	Recommendation	Fixed
8	-	Revise the misleading annotation	Recommendation	Fixed
9	-	Security assumption on external dependencies	Note	-
10	-	Poseidon preimage ownership model	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Compilation errors

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The Circom 2 circuits contain several invalid declarations.

1. In template `MerkleTreeChecker`, signals `left` and `right` are declared inside a `for` loop. However, Circom requires signals to be declared in the template's initial scope, triggering compilation error `T2011`.
2. In circuits `Transfer` and `Unshield`, `var TREE_LEVELS = 20` is declared at the file top level. However, a `var` declaration is not allowed at the file top level, triggering compilation error `P1009`.

```
35      hashers[i] = Poseidon2();
36
37      // Multiplexer for selecting order
38      signal left;
```

```
39      signal right;
40
41      // left = pathIndex ? pathElement : levelHash
42      // right = pathIndex ? levelHash : pathElement
43      left <== pathIndices[i] * (pathElements[i] - levelHashes[i]) + levelHashes[i];
44      right <== pathIndices[i] * (levelHashes[i] - pathElements[i]) + pathElements[i];
45
46      hashers[i].in[0] <== left;
47      hashers[i].in[1] <== right;
48
49      levelHashes[i + 1] <== hashers[i].out;
```

Listing 2.1: circuits/lib/merkle.circom

```
34var TREE_LEVELS = 20;
```

Listing 2.2: circuits/core/transfer.circom

```
36var TREE_LEVELS = 20;
```

Listing 2.3: circuits/core/unshield.circom

Impact The affected circuits cannot be compiled, so no valid artifacts (`.r1cs`, final `.zkey`, or Solidity verifier) can be produced from these sources.

Suggestion Fix the invalid declarations.

2.1.2 Incorrect range in template `AmountRangeCheck`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the circuit `Shield`, the public input `amount` is validated by the template `AmountRangeCheck`, which restricts `amount` to 64 bits. However, this check is overly strict and may block normal transactions. For an asset with 18 decimals, the maximum valid circuit amount is approximately 18.44 tokens, which is insufficient for normal transfers. As a result, legitimate operations above this bound cannot produce a valid proof.

```
31template Shield() {
32   // Public inputs
33   signal input commitment;
34
35   // Private inputs
36   signal input amount;
37   signal input tokenId;
38   signal input owner;
39   signal input blinding;
40
41   // Verify commitment is correctly formed
42   component noteCommitment = NoteCommitment();
43   noteCommitment.amount <== amount;
44   noteCommitment.tokenId <== tokenId;
```

```
45    noteCommitment.owner <== owner;
46    noteCommitment.blinding <== blinding;
47
48    // Enforce commitment matches
49    noteCommitment.commitment === commitment;
50
51    // Verify amount is in valid range
52    component amountCheck = AmountRangeCheck();
53    amountCheck.amount <== amount;
54}
```

Listing 2.4: circuits/core/shield.circom

```
112/**
113 * Amount Range Check
114 * Validates that amount is a valid 64-bit value
115 */
116template AmountRangeCheck() {
117    signal input amount;
118    signal output valid;
119
120    component rangeCheck = RangeCheck(64);
121    rangeCheck.value <== amount;
122
123    valid <== rangeCheck.valid;
124}
```

Listing 2.5: circuits/lib/commitment.circom

Impact Valid deposits and withdrawals cannot produce proofs, breaking normal protocol usage.

Suggestion Enlarge the valid range of `amount` to 254 bits to cover values that may appear on-chain as `uint256`.

2.1.3 Lack of binding on withdrawal signals

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the circuit `Unshield`, the withdrawal proof is expected to bind the publicly declared execution intent. However, the public input `recipient` does not participate in any constraint. In addition, `relayer` does not appear in the circuit statement, even though `Shield.withdraw()` pays `_fee` to the caller-supplied `_relayer`. An attacker can therefore observe a pending withdrawal transaction and front-run it by resubmitting the same proof with `_recipient` and `_relayer` replaced by the attacker's addresses. This redirects both the withdrawn funds and the relayer fee to the attacker.

```
38template Unshield(levels) {
39    // ============ Public Inputs ============
40    signal input root;
```

```
41    signal input nullifierHash;
42    signal input recipient;
```

Listing 2.6: circuits/core/unshield.circom

```
265         // Transfer to recipient
266         (bool success1, ) = payable(_recipient).call{value: netAmount}("");
267         require(success1, "Shield: native transfer failed");
268
269         // Transfer relayer fee
270         if (_fee > 0 && _relayer != address(0)) {
271             (bool success2, ) = payable(_relayer).call{value: _fee}("");
272             require(success2, "Shield: relayer fee transfer failed");
273         }
```

Listing 2.7: atoshi-privacy-contracts/contracts/core/Shield.sol

Impact Attackers can front-run a withdrawal transaction to steal both the withdrawn funds and the relayer fee.

Suggestion Add `relayer` as a public input and constrain both `recipient` and `relayer` in the circuit.

2.1.4 Lack of necessary public inputs

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the circuit `Shield`, only `commitment` is exposed as a public input, while `amount`, `tokenId`, `owner`, and `blinding` are private inputs. The verifier can therefore confirm only that the commitment was computed from some private tuple. The contract cannot verify that the committed `amount` and `tokenId` match the asset actually deposited in the transaction.

```
31template Shield() {
32    // Public inputs
33    signal input commitment;
34
35    // Private inputs
36    signal input amount;
37    signal input tokenId;
38    signal input owner;
39    signal input blinding;
```

Listing 2.8: circuits/core/shield.circom

```
56component main {public [commitment]} = Shield();
```

Listing 2.9: circuits/core/shield.circom

Impact Commitments can encode an amount or token that differs from the actual deposit.

Suggestion Expose `amount` and `tokenId` as public inputs.

2.1.5 Lack of range constraint for signal `fee`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The circuit `Unshield` should ensure that `fee` is a valid amount and does not exceed the withdrawn amount. However, the circuit does not restrict `fee` to the valid amount range. Instead, it checks only whether `amount - fee` falls within the expected range. Because circuit arithmetic operates over a finite field, an out-of-range `fee` can cause the subtraction to wrap around to a small value that passes the range check.

```
91   // ============ Step 6: Validate Fee ============
92   // Fee must be less than or equal to amount
93   signal feeValid;
94   feeValid <== amount - fee;
95   // This will fail if fee > amount (underflow in field)
96   component feeCheck = AmountRangeCheck();
97   feeCheck.amount <== feeValid;
```

Listing 2.10: circuits/core/unshield.circom

Impact A malicious prover may generate a valid proof using an invalid `fee`, potentially causing incorrect fee or withdrawal accounting.

Suggestion Apply a range check to `fee`.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The following unused code should be removed to improve readability and avoid misuse.

1. In template `MerkleTreeChecker`, component `mux` is declared but never used.
2. Template `LeafIndexToPathIndices` appears unused and assigns `pathIndices` through unconstrained `<-` operations.

```
22   component mux[levels];
```

Listing 2.11: circuits/lib/merkle.circom

```
109      pathIndices[i] <-- remaining % 2;
```

Listing 2.12: circuits/lib/merkle.circom

Suggestion Remove the unused code.

2.2.2 Clarify `tokenId` semantics in note commitments

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In template `NoteCommitment`, `tokenId` binds each note to an asset. The implementation uses the token contract address cast to `uint256`, while contract `TokenRegistry` also assigns separate numeric IDs for metadata. The documentation is ambiguous unless these two meanings are explicitly distinguished.

```
10 * - tokenId: Token identifier (uint256)
```

Listing 2.13: circuits/lib/commitment.circom

Suggestion Document that note commitments bind to the token address.

2.2.3 Revise the misleading annotation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In template `DerivePublicKey`, `publicKey` binds note ownership to a user's private key. The implementation defines `publicKey` as `Poseidon(privateKey)`, which is a hash-based identity value rather than a conventional cryptographic public key. However, the annotation states that the template derives a public key and suggests using EdDSA or BabyJubJub in production, making the intended ownership model unclear.

```
62 * Derives public key from private key
63 *
64 * For simplicity, we use: pubKey = Poseidon(privateKey)
65 * In production, consider using EdDSA or BabyJubJub
```

Listing 2.14: circuits/lib/nullifier.circom

Suggestion Revise the annotation to describe the implemented preimage-based ownership model accurately.

2.3 Note

2.3.1 Security assumption on external dependencies

Introduced by `Version 1`

Description This audit assumes the security of all external dependencies, including the underlying cryptographic primitives, the correctness of external Circom libraries, and the proper execution of the trusted setup ceremony used to generate the Groth16 proving and verification keys. The conclusions of this assessment depend on the integrity of these external components and procedures.

2.3.2 Poseidon preimage ownership model

Introduced by `Version 1`

Description In the circuits `Unshield` and `Transfer`, `DerivePublicKey` reconstructs the note owner from a private witness. The implementation defines `owner` as `Poseidon(privateKey)`, which forms a coherent preimage-based ownership model but is not a conventional public-key scheme. Under this model, the preimage is a private key derived from a fixed signature, so security depends on the confidentiality of that signature-derived secret.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS